RIGOROUS CODE GENERATION FOR DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

ALI ALMOHAMMAD

PhD

2013

RIGOROUS CODE GENERATION FOR DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

ALI ALMOHAMMAD

A thesis submitted in partial fulfilment
of the requirements of the
University of Northumbria at Newcastle
for the degree of
Doctor of Philosophy

Research undertaken in the Faculty of
Engineering and Environment

September 2013

This thesis is dedicated to Khuzama Mobark and Isaa Almohammad.

ABSTRACT

This thesis addresses the problem of generating executable code for distributed embedded systems in which computing nodes communicate using the Controller Area Network (CAN). CAN is the dominant network in automotive and factory control systems and is becoming increasingly popular in robotic, medical and avionics applications. The requirements for functional and temporal reliability in these domains are often stringent, and testing alone may not offer the required level of confidence that systems satisfy their specifications. Consequently, there has been considerable research interest in additional techniques for reasoning about the behaviour of CAN-based systems. This thesis proposes a novel approach in which system behaviour is specified in a high-level language that is syntactically similar to Esterel but which is given a formal semantics by translation to bCANDLE, an asynchronous process calculus. The work developed here shows that bCANDLE systems can be translated automatically, via a common intermediate net representation, not only into executable C code but also into timed automaton models that can be used in the formal verification of a wide range of functional and temporal properties. A rigorous argument is presented that, for any system expressed in the high-level language, its timed automaton model is a conservative approximation of the executable C code, given certain well-defined assumptions about system components. It is shown that an off-the-shelf model-checker (UPPAAL) can be used to verify system properties with a high-level of confidence that those properties will be exhibited by the executable code. The approach is evaluated by applying it to four representative case studies. Our results show that, for small to medium-sized systems, the generated code is sufficiently efficient for execution on typical hardware and the generated timed automaton model is sufficiently small for analysis within reasonable time and memory constraints.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and thanks to my supervisors Dr David Kendall and Dr William Henderson. I have been fortunate to do this work supervised by such an excellent team. I should acknowledge that without their generous support, this work would not have been completed.

I would like to thank my colleagues in Lab F7, Pandon Building, School of Computing, for the friendly and kindly atmosphere provided.

I would like also to thank my scholarship sponsor, the ministry of higher education, Syria, for the financial support of my study.

I am grateful, last but not least, to my family, without their loving support and encouragement, I would have given up long ago.

DECLARATION

I declare that the work contained in this thesis has not been submitted for any other award and that it is all my own work. I also confirm that this work fully acknowledges opinions, ideas and contributions from the work of others. Any ethical clearance for the research presented in this thesis has been approved. Approval has been sought and granted by the School Ethics Committee on 17th December 2008. I also confirm that the thesis is (40,725) words in length.

Name: Ali Almohammad

Signature:

Date:

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION AND OVERVIEW

1.1 Embedded System

An embedded system is a computer system that is part of a larger system to perform a dedicated function such as monitoring and controlling. We see such computer systems everywhere around us, for example: cell phones, domestic appliances, medical systems, traffic control systems, and automotive applications. Embedded computer systems are becoming more attached to our life, therefore embedded development methods are very important in order to ensure reliability particularly where a failure may cause loss of life or financial damage. On June 4, 1996 the Ariane 5 rocket launched by the European Space Agency exploded just 40 seconds after initiation of the flight sequence (Dowson, 1997). A software failure was identified as a primary cause of the disaster. The catastrophe was valued at approximately $370 million. More recently, Toyota, the world's largest automobile manufacturer, announced in 2010 the recall of thousands of cars because of a problem in a braking system. A software glitch also has been suspected in the braking system. Embedded system engineers need to ensure that the systems which they deliver will behave correctly. Tools and techniques that support that are highly demanded.

1.2 Real-time Embedded System

Mostly, embedded systems are real-time systems. In other words, they have real-time constraints where the correctness of their behaviour depends not only

on the logical results of the computation, but also on the time at which these results are produced. In a vehicle air-bag example, once a crash is detected, the air-bag has to inflate rapidly within a short time. In this example, the system should respond to the event of the crash on the correct time in order to prevent driver from striking the steering wheel or window. Real-time systems are classified into two types depending on how strict are the timing requirements: soft real-time systems and hard real-time systems (Liu, 2000). A soft real-time system has more relaxed timing constraints. The system can continue to operate even if it fails to meet its deadline. Examples are multimedia applications, telephone switches, and on-line reservation systems. In a hard real-time system, if a timing constraint or deadline is not met, errors consequences may occur threatening human lives or causing sever damage or financial loss. Examples include safety-critical applications: medical machines, automotive and avionics.

Consider the simple example shown in Fig. 1.1 (Kopetz, 1997), where the computer performs a single activity. The system controls the flow of a liquid through a pipe. For a given set-point, the computer system must maintain the flow of the liquid despite changing environmental conditions, such as varying level of the liquid in the vessel or temperature sensitive viscosity of the liquid. The computer system continuously observes the rate of flow, using the flow sensor, and adjusts the control valve accordingly. The response to a change in the flow must occur within a finite period of time in order to prevent an overload situation. This however may require complex calculation in order to obtain the new valve position. Checking that the system meets some functional properties is necessary, for example: whenever an increase of the flow rate is detected, the valve is eventually adjusted. However, verifying non-functional (or timing) properties is very important, for example: whenever an increase of the flow rate is detected, the valve is eventually adjusted before x time units. Tools and techniques that provide a prior analysis about the worst-case behaviour of such systems, are demanded.

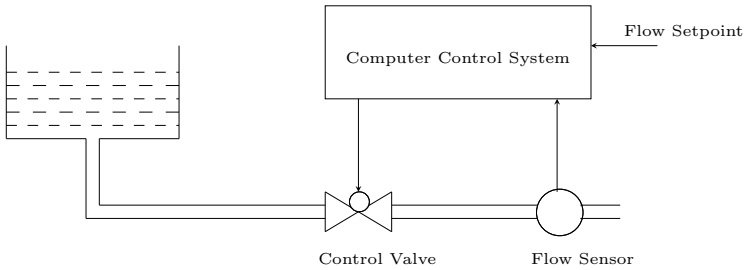


Fig. 1.1: Flow Regulator System.

1.3 Distributed Embedded System

Embedded systems tend to be distributed because of the nature of the environment in which they operate. In this case a system may have a number of computing nodes that are interconnected by a communication network in order to exchange information for the purpose of monitoring and control. Modern vehicles may have over a hundred computing units to control, for example, air-bag, driver's doors, anti-lock brakes, engine functions and many other activities in the car (Pop et al., 2004). Distribution is required for various reasons such as performance increase, location of sensors and actuators, and fault tolerance (Caspi et al., 1999). For instance, consider the example shown in Fig 1.1. For a geographical reason, the computing node that reads the flow rate, may be placed close to the flow sensor and interconnected with a suitable communication bus with another computing node that controls the valve. Unfortunately, it is a challenging task to design and implement real-time embedded systems in such a way that guarantees that the functional and timing properties are satisfied under all possible workloads. The problem becomes even harder when the system is distributed.

A lot of distributed embedded systems are implemented using Controller Area Network (CAN). The CAN is the dominant network in automotive and factory control systems and is becoming increasingly popular in robotic, medical and avionics applications. In the following, a brief introduction of the CAN is

presented.

The Controller Area Network

Controller Area Network (CAN) (Bosch, 1991; ISO-CAN, 1993; Natale et al., 2012) is a broadcast, message-oriented communication protocol developed originally for the automotive industry by Bosch GmbH in the mid-eighties. CAN was devised to replace the complex wiring harness in automobiles with a two-wire bus, and suited to operate in a harsh electromagnetic environment at transmission speeds of up to 1 Mbit/s over short distances. CAN is a multi-master protocol, any node on the network can send a broadcast message to other nodes. The message does not contain the address of the destination node(s), but it has a unique static number which defines the priority of the message in the network. CAN implements the carrier-sense, multiple-access (CSMA/CA) protocol with a deterministic collision avoidance policy. This feature has made CAN particularly suitable for hard real-time systems which require high reliability. Currently its use has been expanded to include new application domains, for example: manufacturing, construction, agriculture and healthcare (Ortiz et al., 2011; Eugenio, 2008; Riti and Pozzi, 1999; Parent and Cassin, 1999).

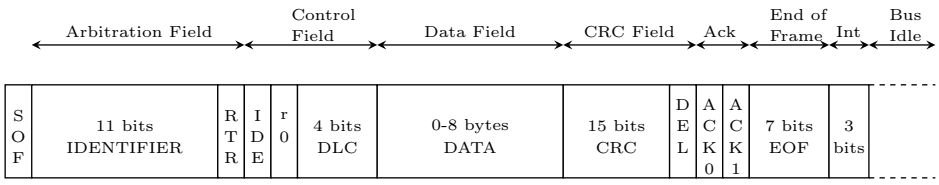


Fig. 1.2: CAN Frame - Standard Format.

Messages are transmitted over the CAN network as fixed format frames which consist of a data field, a message identifier field and other trailing fields, see Fig 1.2. The data field is between 0 and 8 bytes long. The identifier field is 11 (29) bits in the standard (extended) frame format. The other fields of the frame are explained in Table 1.1. Additional bits are inserted when a frame is trans-

Field name	Purpose
SOF	Denotes the start of frame transmission.
ID	Represents message identifier.
RTR	Remote transmission request.
IDE	Identifier for the data which also represents the message priority.
r0	Reserved bit.
DLC	Data length code which represents the number of bytes of data.
DATA	Represents message data field which is between 0 and 8 bytes long.
CRC	Cyclic redundancy check.
DEL	CRC delimiter.
ACK0	ACK slot.
ACK1	ACK delimiter.
EOF	End of frame.
Int	Inter-frame space.

Tab. 1.1: CAN frame fields.

mitted over the network for synchronisation. This process is called *bit stuffing* that means after five consecutive equal bits, a complementary bit is inserted into the bit stream. Bit stuffing occurs from SOF up to, but not including, the CRC delimiter. Stuff bits introduce uncertainty to the transmission time of CAN messages. However, the maximum number of stuff bits can be calculated in order to estimate the worst-case transmission time of a CAN message. This is discussed in detail in section 2.3.4.

The identifier represents the priority of the message which is a non-negative integer starting from 0; the smaller the number the higher the priority. When two or more nodes try to transmit a message, the node with the higher priority message gains access to the bus. First when the bus is free, a number of nodes may start to transmit at the same time. Each node first transmits the message identifier starting from the most significant bit, and then it monitors the bus. In this mechanism, the bit is classified as either *dominant* or *recessive*. The node can only read a recessive bit if all other nodes write recessive bits, otherwise it reads a dominant bit. The behaviour of the bus in this case is similar to an

AND-gate. When a node monitors a bit value that is not the one transmitted it stops transmission, and behaves as a receiver to the highest priority message, and then waits until the bus becomes idle again. Therefore, during arbitration, the node with the highest priority message wins and continues to transmit the rest of the message. Additionally, the message identifier expresses the type of the message. Each node may be configured to accept a subset of messages. A receiver node performs the acceptance test to the received message. If the message is accepted it is stored in a receive buffer, otherwise it is rejected.

1.4 Formal Methods

Embedded systems may have high reliability requirements, e.g. a mean time to failure of 10^9 hours is not unusual. Traditional approaches to testing embedded systems may not alone provide the required confidence in their reliability. In (Gluck and Holzmann, 2002) two problems are identified in conventional testing in terms of concurrent systems: "limited controllability" and "limited observability". The first one means that it is not possible to control the specifics of thread interleavings; the second one means that it is very hard to reproduce the error scenario to identify the root cause. Therefore, finding errors such as race conditions and deadlock in concurrent software is very challenging using conventional testing.

However, these approaches can be supplemented by a variety of analytic techniques. One such technique is model-checking (Baier and Katoen, 2008). Model-checking has proved to be a very effective method to verify requirements and the design of concurrent systems and communication protocols. Basically, a model-checking tool accepts an abstract model of the system and a specification of properties of the system. The tool then performs an exhaustive state-space search to check if the model satisfies the given specification. A counterexample (sequence of events or path) is generated if the specification is not satisfied

by the model. Therefore, by studying the counterexample, the error may be discovered and corrected. A major obstacle to the widespread deployment of model-checking in practice is the state-space explosion problem: the number of states to be checked in a realistic model of the system may exceed the available computing resources. Furthermore, model-checking verifies a system model which is an abstraction of the actual system. Thus the model may exhibit a behaviour that can not be expressed by the actual implementation of the system. Despite of these limitations, model-checking can increase the level of confidence in a system design.

1.5 Related Work

The problem of real-time system has been subject to exhaustive research during the last few decades. Many approaches have been proposed to ensure that systems hold some useful properties. The synchronous approach is based on very conservative assumptions on system computations and communications which make the system difficult to implement particularly when the system is distributed. Traditional scheduling analysis has been widely used by real-time system engineers. It provides a simple mathematical analysis of system behaviour. Based on the success of its application to single-processor systems, the approach has been extended to the distributed applications. However, the approach requires restrictive assumptions on the system implementation, and does not allow system level properties to be checked. Other approaches based on formal languages have been proposed, but they are mainly limited to uni-processor applications.

1.5.1 Synchronous Approach

The synchronous languages Esterel (Berry, 2000), Lustre (Halbwachs et al., 1991) and Signal (LeGuernic et al., 1991) are designed around the *synchrony hypothesis* which assumes a system responds to its environment's events in zero time (Benveniste et al., 2003). Moreover, all communications between the system components are also performed instantaneously. The concept is similar to the synchronous model of digital circuits. The circuits are described using gates that must react during one clock cycle which means conceptually in zero time. This approach allows one to reason formally about the operations of the system (Benveniste and Berry, 1991). Synchronous languages and their compilers are now widely used in industry for automotives, railways, and avionics.

The Esterel language is suited to the development of control-dominated embedded reactive applications (Potop-Butucaru et al., 2007). An Esterel program consists of a collection of concurrently running threads which are described in a traditional imperative syntax. The concurrency however is compiled away in order to generate a single-threaded source program that behaves like a state machine at run-time. Many compilers have been developed for Esterel such as Esterel Technologies Compiler v7 (Esterel-Tech, 2005), Saxo-RT Compiler (Weil et al., 2000; Closse et al., 2002), and Columbia Esterel Compiler (Edwards and Zeng, 2007).

In order to validate the synchronous assumption in realistic applications of a synchronous language, the tool TAXYS (Bertin et al., 2000; Closse et al., 2001) has been developed. The main goal of TAXYS is to generate a formal model that captures the temporal behaviour of a real-time application and its external environment. The formal model is produced in a timed automata form (Alur and Dill, 1994). Esterel is used as a development language for the application. The KRONOS model checker (Daws et al., 1996) is used to check whether the program satisfies its timing constraints. Although the tool is applied success-

fully for some applications, for example (Bertin et al., 2001) and (Tripakis and Yovine, 2001), the approach is limited to a single-task implementation of a synchronous real-time application running on a single-processor platform (Sifakis et al., 2003).

Generating an executable code from a synchronous language for a distributed architecture has been addressed in Next-TTA (Caspi et al., 2003) and COLA (Haberl et al., 2008a,b). The approach of Next-TTA translates a high-level control design of Simulink (MathWorks, 2012) to a SCADE/Lustre program for validation purpose. Then the implementation is derived for TTA execution layer. TTA (Time Triggered Architecture) (Kopetz, 1997) supports distributed implementations based on a synchronous bus. The TTA conforms with a notation of global synchronisation and ideally matches the synchronous assumption. Although the tool aims to benefit from the Lustre model-checker Lesar (Ratel et al., 1991) to check whether the implementation satisfies its functional and timing properties, the analysis is limited to uni-processor implementations (Caspi et al., 2003) because the tool Lesar assumes only this kind of implementation when its input model is constructed. COLA (Kugele et al., 2007) is a component language for design and development embedded systems. The language has a formal semantics based upon synchronous dataflow. An approach is presented in (Haberl et al., 2008a) to translate models given in COLA to C code. The approach employs a time division multiple access (TDMA) communication schema (similar to TTA) to ensure the timely delivery of data in order to retain the synchronous semantics of the COLA model (Haberl et al., 2008a). Furthermore, although COLA is defined by a rigorous formal semantics in which automated tools, such as model-checking, can be applied to check its correctness, such a verification tool does not yet exist for the language, and it remains for future work (Haberl et al., 2008a).

Although time-triggered systems obey well the synchronous approach, main-

taining a global clock for both computation and communication is difficult to implement, and the overhead of the implementation is often large when adopting the fully synchronous approach (Potop-Butucaru and Caillaud, 2007). Asynchronous communication schemes (e.g. CAN (Bosch, 1991)) which are now widely used in industries, allow a number of computing nodes operating at different rates to be connected via a communication bus with no need to a global clock for synchronisation. Globally Asynchronous Locally Synchronous (GALS) is an architecture emerged to combine the two approaches. In this architecture, synchronous components are connected via an asynchronous communication media. The requirement for a global time is removed when a synchronous specification is implemented within the GALS model (Potop-Butucaru and Caillaud, 2007).

1.5.2 Scheduling Analysis Approach

This approach has been exhaustively studied in the real-time systems literature. A real-time application is considered to be composed of a set of *tasks* that interact. A task is a piece of code which is executed in response to an event from the environment. Tasks may share resources such as processor, memory, and communication media. Timing requirements of a system design are represented in a form of periods, deadlines and priorities to the tasks. The main role of the scheduling approach is to provide an analysis used to confirm that the timing constraints of the system are satisfied. There are two main scheduling approaches: the cyclic executive approach and the priority-based approach (Burns and Wellings, 2001). In the former approach, each task has cyclic access to the processor in a predefined order. In the latter approach, each task is assigned a unique priority according to some policy (e.g., RMA or EDF (Burns and Wellings, 2001)). When a higher priority task is released during the execution of a lower priority one, then the processor will imme-

diately switch to execute the instructions of the higher priority task. This is called a preemptive schema. However, in the non-preemptive schema, the lower priority task is allowed to complete its execution before switching to the higher priority task. A simple schedulability analysis is given by Liu and Layland (Liu and Layland, 1973) in order to test if a set of fixed-priority periodic tasks meet their deadlines. An enhanced analysis is proposed in (Joseph and Pandya, 1986) which calculates the worst-case response time of each task, and then compares it with the deadline of the task. Although the main focus of traditional analysis is on the worst-case behaviour in order to ensure that tasks meets their deadlines, the best-case response time analysis has been addressed in (Redell and Sanfridson, 2002) and (Bril et al., 2004). One such application of this analysis is to estimate the maximum variation in a task response time. The scheduling analysis has been extended to allow distributed systems performance evaluation where end-to-end response times are computed for tasks running in a distributed environment and communicate via a real-time communication protocol. For example, the work of Tindell et al (Tindell and Hansson, 1995), Henderson et al (Henderson et al., 1998), and Redell et al (Redell et al., 2004).

The scheduling approach in general assumes very restrictive assumptions on the system implementation in order to analysis the behaviour of the system. For example, all tasks are periodic, tasks have deadlines equal to their periods, and tasks are independent. Additionally, special purpose resources such as a real-time operating system (RTOS) (e.g., (WindRiver, 1999)) and protocols (e.g., (Sha et al., 1990)), are often required to implement preemptive scheduling policies and to avoid deadlocks. Moreover, the approach does not allow system level properties (e.g. safety and functionality) to be verified, and only deals with implementation level properties (tasks meet their deadlines).

1.5.3 Other Approaches

Process algebra languages have been widely used in the specification and design telecommunication protocols and distributed systems. A system is represented as a process, or composed of other smaller processes. Their formal semantics makes them amenable to formal verification (e.g. (Garavel and Sifakis, 1990)) as well as simulation. Their formalism has been extended to allow modelling real-time aspects of a system, for instance: ET-LOTOS (Léonard and Leduc, 1997), ATP (Nicollin and Sifakis, 1994). Despite their expressiveness and clean formalism, and the availability of analysing tools: simulators, model-checkers, and theorem provers, their applications have been limited to support only the specification and verification (e.g. TRAIAN Compiler of LOTOS NT (Garavel et al., 2002)) rather than implementation of a system (and distributed system in particular). Moreover, they can not deal with a broadcast communication mechanism such as CAN (Bosch, 1991; ISO-CAN, 1993) that is most frequently employed in the implementation of distributed embedded systems. However, generating an implementation code from a process algebra language for uni-processor platform has been addressed in the work of Bradly et al (Bradley et al., 1994c,b,a). A real-time system can be represented in the timed process algebra, AORTA (Bradley, 1995). The language is designed to consider both verification and implementation of a system. An implementation C code is generated for each process of an AORTA design from the same graph that is used in the generation of an analytical model (Bradley, 1995). The AORTA system can be validated via simulation and formally verified by model-checking (Bradley et al., 1996).

PTIDES is a programming model for distributed real-time systems (Zhao et al., 2007; Lee et al., 2009). It is based on a discrete-event model which has been used for simulations. In this model, a network of components reacts to input events in time-stamp order and produces output events in time-stamp order. PTIDES

leverages network time synchronisation with bounded error and bounded latency in order to use the model to produce efficient distributed implementations. However, the approach, similarly to traditional scheduling analysis, is limited to schedulability analysis which does not provide system level analysis, and uses a special purpose architecture (PRET architecture (Liu et al., 2010)) to achieve timing predictability.

Additionally, other formalisms such as Time Petri Nets (Berthomieu and Diaz, 1991), Timed Automata (Alur and Dill, 1994), and State Machines have been applied in the development of real-time embedded systems, and implemented in the tools: Roméo (Lime et al., 2009), Times (Amnell et al., 2003) and IAR visualSTATE (IAR-Systems, 2012) respectively. However, in addition to their less expressiveness (i.e. low-level representation format) compared to the process algebra formalism, their usages has not yet considered distributed implementations. Roméo facilitates automated verification via model-checking for the time petri net model of real-time system. The tool does not support the implementation of such systems. Other works however consider this problem, i.e. producing an implementation program (e.g. RT Java) from a Petri net model for a real-time system, see for example (Moreno et al., 2006). In Times, an approach to modelling and implementing embedded systems that combines both schedulability analysis and formal verification is presented in (Norstrom et al., 1999). The idea is to extend the standard time automata (Alur and Dill, 1994) with real-time tasks to allow a more relaxed task model (e.g. non-periodic tasks) to be analysed using a formal verification tool such as UPPAAL model checker (Behrmann et al., 2004). The approach is implemented in the tool, Times (Amnell et al., 2003). Although an executable code can be generated from the extended timed automata model, the approach assumes that the generated code is executed on a uni-processor platform that guarantees the synchronous hypothesis (Amnell et al., 2003). A state machine-based approach (such as (Samek, 2008) and (IAR-Systems, 2012)) generates event-driven

code for uni-processor embedded systems. The tool IAR visualSTATE (IAR-Systems, 2012) can verify a fixed list of untimed properties such as the absence of deadlocks and unreachable states. More complex properties can be verified by constructing another state machine, that expresses the desired property, parallel to the system design.

1.6 The Dissertation

1.6.1 Justification

The work presented in this thesis addresses the problem of generating executable code for CAN-based distributed embedded systems in a way that guarantees that both functional and timing properties expressed in a high-level formal language are satisfied. The thesis proposes a novel approach in which system behaviour is specified in CANDLE, a high-level language which is given a formal semantics by translation to bCANDLE, an asynchronous process calculus. A bCANDLE system is translated automatically, via a common intermediate net representation, both into executable C code and into a timed automaton model that can be used in the formal verification of a wide range of functional and temporal properties.

1.6.2 Structure and Contributions

Chapter 2 introduces our code and model generation approach, and provides all the essential details including models, languages, and formal notations. The chapter presents no new results but provides the necessary information on which the rest of the thesis is built.

Chapter 3 is concerned with the implementation of the formal language. The executable code is derived from the language via an intermediate model. An

efficient C representation of the intermediate model is presented. The chapter presents our implementation model which defines certain assumptions about system components. A single broadcast asynchronous communication mechanism is adopted and implemented. The communication mechanism is an abstraction of the CAN. All communications occur through this mechanism and never through the use of shared variables. This single notion, employed both for external and local communication between system components, provides flexibility to the system developer to freely distribute system components on a number of nodes, and simplifies the process of generating a formal model of the system. An AADL-like language is adopted to describe the system architecture. The Architecture Analysis and Design Language (AADL) (Feiler et al., 2006) is an industry standard language to specify a system architecture. The system description provides details to the code generator about processes, nodes, process-to-node allocation, scheduling algorithm, tick rate, and communication details, including the IDs of messages and network transmission rate.

Chapter 4 presents a rigorous argument that, for any system expressed in the high-level language, its formal model is a conservative approximation of the executable C code. This allows the system developer to conclude that if a model satisfies any universally quantified property, then it is guaranteed that the implementation will also satisfy the same property.

Chapter 5 proposes a number of methods that ensure an atomic update of data which is required to implement the semantics of the language correctly. The methods are evaluated against some criteria we identify depending on the worst-case behaviour analysis of the methods in order to select a suitable one for our code generation approach.

Chapter 6 assesses the applicability and performance of the approach by implementing four case studies. The performance is measured in terms of the computational effort required to generate an executable code and a formal model

for a given design, and the computational resources including memory (RAM and ROM) and CPU load required to execute the examples on the target. Performance results are compared with the results obtained from an alternative method employing a widely-used real-time kernel that implements the same case studies. The chapter also assesses the tractability of the formal models which are generated from the case studies. A number of functional and temporal properties are verified using an off-the-shelf model checker.

Chapter 7 summarises the work, discusses the limitations of the work, and suggests possible directions for future research.

2. METHODS, TECHNIQUES AND TOOLS

This chapter introduces our code and model generation approach, and provides all the essential details including models, languages, and formal notations, on which the rest of the thesis is built.

2.1 Overview of the Approach

An overview of our code and model generation approach is depicted in Fig. 2.1. The primary component in our approach is bCANDLE (Kendall, 2001b), a timed process calculus intended for modelling CAN-based embedded systems. The language features a value-passing, broadcast communication primitive, message priorities and an explicit time construct. The motivation to use bCANDLE is that it enables system developers to construct system models that are amenable to formal analysis by model-checking. There is a well-defined translation from bCANDLE to timed automata (TA) models upon which standard model checkers can be used. The existing translator is built upon a low-level intermediate net representation (Kendall, 2001b). The idea of the thesis is to make use of the net representation developed by the TA translator to generate executable code. The approach of generating executable code from a net is discussed in Chapter 3. The main reason for employing the net is that it will be easier to establish a connection between the behaviour of the model and the executable code since they are both produced from the same source.

This will be the subject of Chapter 4 of the thesis. The executable code is generated in the C language. The model is generated in the form of a timed automaton so an off-the-shelf model-checker (such as UPPAAL) can be used to verify system properties. In order to be able to verify temporal properties of the system, time bounds of the execution of system components can be predicted using static analysis tools such as the *Bound-T* (Tidorum, 2012) and *aiT* (Absint, 2012). Although bCANDLE is a simple language, it is likely to be too low level for routine use in system description. Therefore, CANDLE (Kendall, 2001b) was introduced for the purpose of a system design. The CANDLE is a high-level programming language dedicated for CAN-based embedded systems. The language has a formal semantics defined by translation into bCANDLE. In summary, the system is expected to be designed in CANDLE. Then, a bCANDLE model is produced from the CANDLE design of the system. Next, a net representation is derived from the bCANDLE model. Finally, an executable C code of the system is generated from the net.

The rest of the chapter is organised as follows. Section 2.2 presents the main assumptions made and constraints on systems for which the code and model generation approach is proposed. Section 2.3 introduces the bCANDLE modelling language and its formal semantics. Section 2.4 introduces the intermediate representation of bCANDLE, the net and its formal semantics. Section 2.5 outlines the translation rules of bCANDLE into the net. Section 2.6 introduces the high-level language of bCANDLE, CANDLE. Section 2.7 outlines the translation rules of CANDLE into bCANDLE. Finally, section 2.8 concludes the chapter.

2.2 System Characteristics

The proposed code and model generation approach targets a class of embedded systems (Fig. 2.2) characterised by a number of properties:

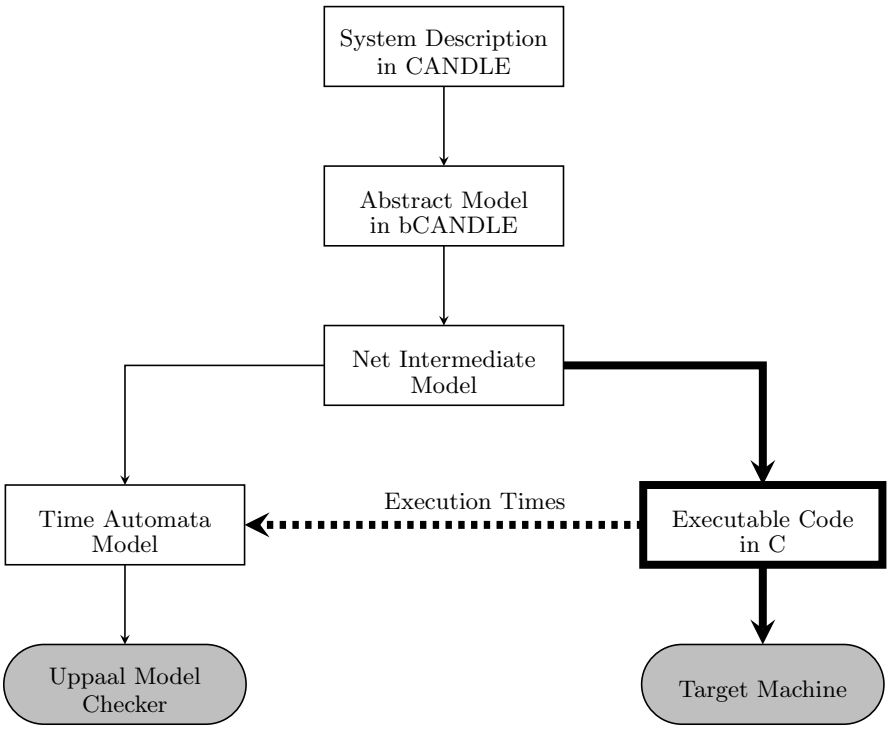


Fig. 2.1: Overview of Code and Model Generation Approach.
Work done as part of this thesis is shown in bold.

1. A system comprises a number of software processes that are statically distributed to computing nodes.

2. A computing node consists of a processor which has access to a local memory, a programmable timer, zero or more communication controllers, and zero or more sensors and actuators to interact with the physical world.

3. A restricted scheduling approach (e.g., cooperative or round-robin, discussed in section 3.2.3) is applied when two or more processes are allocated to a single computing node in order to allow off-line calculation of computation response times.

4. Processes communicate via asynchronous broadcast channels which implement an abstraction of the CAN protocol in which the send operation is non-blocking and the receive operation is blocking. The abstraction of the CAN is formed by the following assumptions:

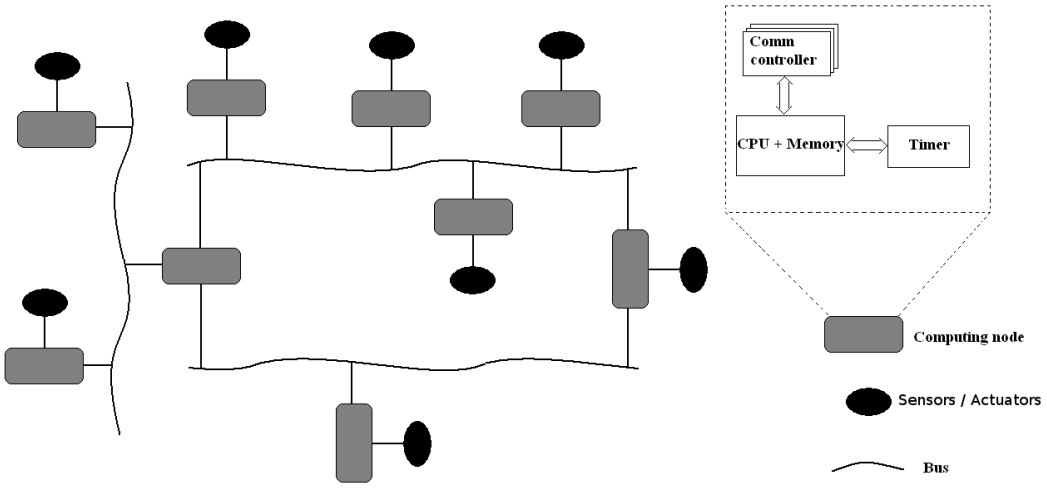


Fig. 2.2: Distributed Embedded System (Kendall, 2001b).

- communication channels operate without errors,
- the details of bit-level data transmission are abstracted so that messages are assumed to be transmitted atomically,
- the CAN message ID field is 11 bits in length.

5. Processes that reside in the same computing node are not allowed to share memory and typically use (local) broadcast channels to communicate.

6. Shared access to I/O is not allowed, i.e. in the case of multi-tasking, access to a sensor or actuator is limited to a single software process.

2.3 The bCANDLE Modelling Language

bCANDLE (Kendall et al., 1997, 1998b; Kendall, 2001b) is a timed process calculus dedicated for modelling distributed, real-time embedded systems based on the CAN network. The language features a value-passing, broadcast communi-

cation primitive, message priorities and an explicit time construct. A bCANDLE system comprises three components: a *data model*, a *network model*, and a *process model*. It is defined formally by a tuple (P, N, D). In the following, the definition and notations are presented for each component of bCANDLE system.

2.3.1 Data Model

Let Var be a finite set of data variables. Each variable $x \in Var$ takes its value from some non-empty, finite set of values, $\mathsf{type}(x) \subset V$, where V is the set of data values. We assume that V contains at least the distinguished value $\perp$, where $\perp \notin \bigcup_{x \in Var} \mathsf{type}(x)$, which is taken to be the "undefined" data value, then:

- $Valuation \mathrel{\widehat{=}} Var \to V$
- $Operation \mathrel{\widehat{=}} Valuation \leftrightarrow Valuation$
- $Predicate \mathrel{\widehat{=}} 2^{Valuation}$

Let Ω be a finite set of operation names, Γ be a finite set of predicate names, then a *data environment* D over Var, Ω and Γ is a tuple $D = (\mathsf{type}, \mathsf{operation}, \mathsf{predicate}, \mathsf{val})$ where

- $\mathsf{type} : Var \to 2^V$ is a total function, giving for each variable $x \in Var$, a non-empty, finite set of data values, $\mathsf{type}(x)$, ranged over by x;
- $\mathsf{operation} : \Omega \to Operation$ is a total function, giving for each operation name $\omega \in \Omega$, an operation, $\mathsf{operation}(\omega)$, which interprets it;
- $\mathsf{predicate} : \Gamma \to Predicate$ is a total function, giving for each predicate name $\gamma \in \Gamma$, a predicate, $\mathsf{predicate}(\gamma)$, which interprets it;

- $\mathsf{val} : \mathit{Var} \to V$ is a total function which, for each variable $x \in \mathit{Var}$, gives the current valuation of x, where $\mathsf{val}(x) \in \mathsf{type}(x)$ or $\mathsf{val}(x) = \perp$.

Let $D = (\mathsf{type}, \mathsf{operation}, \mathsf{predicate}, \mathsf{val})$ be a data environment. Let $x, y \in \mathit{Var}$ be data variables, and $v \in V$ a data value, then the following notational conventions are employed:

- $D.\,\mathsf{type}$, $D.\,\mathsf{operation}$, $D.\,\mathsf{predicate}$ and $D.\mathsf{val}$ denote type, $\mathsf{operation}$, $\mathsf{predicate}$ and val, respectively.
- $D.x$ denotes the value $\mathsf{val}(x)$.
- $D[x := v]$ denotes the data environment $D' = (\mathsf{type}, \mathsf{operation}, \mathsf{predicate}, \mathsf{val}')$ where $\mathsf{val}'(x) = v$ and $\mathsf{val}'(y) = \mathsf{val}(y)$ for all $y \not\equiv x$ ($\equiv$ denotes syntactic identity and $\not\equiv$ its negation).
- $D \xrightarrow{\omega}_{\mathsf{d}} D'$ abbreviates the condition $(\mathsf{val}, \mathsf{val}') \in \mathsf{operation}(\omega) \wedge D' = (\mathsf{type}, \mathsf{operation}, \mathsf{predicate}, \mathsf{val}')$. The operation name ID is reserved for the identity relation on valuations, i.e. it must be interpreted in any data environment by the operation $\mathsf{operation}(\mathit{ID}) \triangleq \{(\mathsf{val}, \mathsf{val}) | \mathsf{val} \in \mathit{Valuation}\}$
- $D \models \gamma$ abbreviates the condition $\mathsf{val} \in \mathsf{predicate}(\gamma)$.

2.3.2 Network Model

The network model of bCANDLE is an abstraction of the CAN network which consists of a number of broadcast channels. Each channel implements an abstraction of the CAN protocol. A message transmitted through a channel is viewed as a pair consisting of a data value and an identifier. The data value and the identifier corresponds to the data field and the arbitration field of the CAN frame respectively. Transmission of a message is divided into three phases: *pre-acceptance*, *acceptance*, and *post-acceptance* phase. The acceptance phase is the interval during the transmission of a message when receiver nodes perform

their acceptance test. The pre-acceptance phase extends from the beginning of the transmission to the point of acceptance. The post-acceptance phase extends from the the point of acceptance to the point at which the channel next becomes free. In practice, the location of the acceptance point may vary from one type of CAN controller to another. For example, for some CAN controllers, the acceptance point may occur on the leading edge of the ACK0 bit of the CAN frame, see Fig 1.2. It is assumed that the time which passes during the pre-acceptance and post-acceptance phases can be calculated for all messages. This time is called *transmission latency*. The transmission latency of a message gives the upper and lower bounds on the time which passes during the pre-acceptance and post-acceptance phases of the message.

A channel is defined by the tuple $(M, \prec, \delta, s, u)$ where:

- $M \subseteq I \times V$ is a set of messages. I is a set of message identifiers and V is a set of data values.
- $\prec$: $I \leftrightarrow I$ is a priority ordering of messages.
- $\delta : M \rightarrow \mathbb{R}_\infty \times \mathbb{R}_\infty \times \mathbb{R}_\infty \times \mathbb{R}_\infty$ is a transmission latency function. The functions $\delta^{lb}, \delta^{ub}, \delta^{lB}, \delta^{uB} : M \rightarrow \mathbb{R}_\infty$ give the lower and upper bounds on the duration of the pre- and post-acceptance phases for the transmission of a message. δ^{lb} (resp. δ^{ub}, δ^{lB}, δ^{uB}) is abbreviated as lb (resp. ub, lB, uB).
- s is a transmission status. The channel is either *free* or is transmitting a message (pre-acceptance, acceptance or post-acceptance phase). The notation shown in Table 2.1 denotes the transmission status.
- u is a message queue.

It is assumed that M, $\prec$ and δ are static components, they are defined at the initialisation of a system and are unchanged after that. By contrast, s and u are

Notation	Transmission Status
$\downarrow$	FREE, no message is in transmission.
$\overset{t1,t2}{\rightsquigarrow} m$	pre-acceptance phase of transmission of message m with bounds t_1,t_2 on time to completion, $0 \leq t_1 \leq \mathsf{lb}$, $0 \leq t_2 \leq \mathsf{ub}$.
$\uparrow m$	acceptance point in transmission of m.
$m \overset{t_1,t_2}{\rightsquigarrow}$	post-acceptance phase of transmission of message m with bounds t_1,t_2 on time to completion, $0 \leq t_1 \leq \mathsf{lB}$, $0 \leq t_2 \leq \mathsf{uB}$.

Tab. 2.1: Transmission Status Notation.

dynamic components which are used to model the current transmission status and message queue as a system evolves.

The message queue modelled here represents a single shared queue for a whole network of nodes that are communicating using the same channel. It is assumed that a transmitting node only attempts to transmit its highest priority message. Additionally, a node that has a number of pending messages always attempts to transmit the highest priority message as soon as the channel becomes free. This implies that the channel can not become free between the transmission of messages if there are pending messages. It means that the CAN controller will arbitrate for the bus immediately after sending the previous message, and will only release the bus in case of lost arbitration. This is important to ensure that the transmission of a pending message is not deferred by beginning transmission of a lower priority message. Each CAN controller must have a suitable buffer management mechanism to respect these assumptions. This ideal behaviour of CAN was identified by Tindell et al (Tindell and Burns, 1994; Tindell et al., 1994) for scheduling analysis of CAN network. If all nodes follow the same protocol to transmit messages, then the internal queue of all nodes can be viewed as one large queue for a whole network in which messages are placed in priority order, assuming that each message is assigned a unique priority number in a network.

A bCANDLE network is a set of channels in which each channel is given a unique identifier. Let K be a set of channel identifiers, then a network N over

K is an indexed set of channels which is expressed as follows:

$$N = ((M, \prec, \delta, s, u)_k \mid k \in K)$$

Let c be a channel, then the notation $N[k := c]$ denotes the network N', where $N'_k = c$ and $N'_{k'} = N_{k'}$ for all $k' \in K \setminus \{k\}$.

A channel c can act independently, by performing a discrete change in its transmission status or its message queue, to become c' which gives a new network $N' = N[k := c']$. Alternatively, the state of of the whole network may be affected as time progresses. The relation $N \xrightarrow{\lambda_{\mathsf{nt}}}_{\mathsf{n}} N'$ represents a change of state from N to N' annotated with the label λ_{nt} which ranges over $A_{\mathsf{n}} \cup \mathbb{R}$. A_{n} is a set of network action labels used to annotate discrete state changes. Elements of $\mathbb{R}$ are used to annotate state changes due to the passage of time.

Fig. 2.3 gives the network transition rules. The rules are expressed using the structural operational semantics (SOS) (Plotkin, 2004; Nielson and Nielson, 1991) style. SOS is the predominant approach for giving a meaning to programming and specification languages. It generates a labelled transition system, whose states are the terms of the language, and whose transitions between states are obtained inductively from a collection of transition rules of the form $\frac{\texttt{premises}}{\texttt{conclusion}}$. The validity of the premises of a transition rule, under a certain substitution, implies the validity of the conclusion of this rule under the same substitution (Aceto et al., 2001). The rules **N.1**, **N.2**, **N.3** and **N.4** give the transition rules due to discrete state changes of a network, whereas the rule **N.5** gives the transition rule due to a time progress of a network. The function $\mathsf{tcp}(N)$ denotes the maximum time progress allowed for N.

$$\textbf{N.1}\quad \frac{N_k = (\downarrow, m : u)}{N \xrightarrow{k \rightsquigarrow m}_{\mathsf{n}} N[k := (\overset{\mathsf{lb,ub}}{\rightsquigarrow}\ m, u)]}$$

$$\textbf{N.2}\quad \frac{N_k = (\overset{0,\_}{\rightsquigarrow}\ m, u)}{N \xrightarrow{k \uparrow m}_{\mathsf{n}} N[k := (\uparrow m, u)]}$$

$$\textbf{N.3}\quad \frac{N_k = (\uparrow m, u)}{N \xrightarrow{m \rightsquigarrow k}_{\mathsf{n}} N[k := (m\ \overset{\mathsf{lB,uB}}{\rightsquigarrow}, u)]}$$

$$\textbf{N.4}\quad \frac{N_k = (\_\ \overset{0,\_}{\rightsquigarrow}, u)}{N \xrightarrow{k \downarrow}_{\mathsf{n}} N[k := (\downarrow, u)]}$$

$$\textbf{N.5}\quad \frac{0 \leq t \leq \mathsf{tcp}(N)}{N \xrightarrow{t}_{\mathsf{n}} N + t}$$

Fig. 2.3: Network Transition Rules.

Example of network behaviour

Assume a network that consists of a single channel which can transmit messages of type *flow*. The transmitted values of the flow sensor reading are abstracted, where 0 represents a reading at the low level, and 1 represents a reading at the high level. The network then can be defined as follows:

$$N = \{k \mapsto (M, \prec, \delta, \downarrow, \langle flow.1 \rangle)\}$$

Where $I = \{flow\}$, $V = \{0, 1\}$ and $M = I \times V$. As there is a single channel, then $K = \{k\}$. The function δ gives the transmission latencies in μs, as follows:

	$flow.\_$
δ^{lb}	86
δ^{ub}	106
δ^{lB}	24
δ^{uB}	24

In the table, $flow.\_$ could be $flow.0$ or $flow.1$. It is assumed that the message

$flow.1$ has been already placed in the message queue; a possible trace of the network behaviour is shown in Fig. 2.4. The trace starts from the initial state $(\downarrow, \langle flow.1 \rangle)$ in which the channel is free and the message is queued. Then, the time behaviour of the network progresses following the two-phase model. The network performs a discrete action first using one of the rules (N.1, N.2, N.3, and N.4) and then a time elapses using the rule (N.5) until the channel becomes free and the message queue becomes empty. After that, the network may progress using time transitions.

$$
\begin{array}{lll}
(\downarrow, \langle flow.1 \rangle) & \xrightarrow{k \rightsquigarrow flow.1}_{\mathsf{n}} & (N.1) \\
(\overset{86,106}{\rightsquigarrow} flow.1, \langle\rangle) & \xrightarrow{100}_{\mathsf{n}} & (N.5) \\
(\overset{0,6}{\rightsquigarrow} flow.1, \langle\rangle) & \xrightarrow{k \uparrow flow.1}_{\mathsf{n}} & (N.2) \\
(\uparrow flow.1, \langle\rangle) & \xrightarrow{0}_{\mathsf{n}} & (N.5) \\
(\uparrow flow.1, \langle\rangle) & \xrightarrow{flow.1 \rightsquigarrow k}_{\mathsf{n}} & (N.3) \\
(flow.1 \overset{24,24}{\rightsquigarrow}, \langle\rangle) & \xrightarrow{24}_{\mathsf{n}} & (N.5) \\
(flow.1 \overset{0,0}{\rightsquigarrow}, \langle\rangle) & \xrightarrow{k \downarrow}_{\mathsf{n}} & (N.4) \\
(\downarrow, \langle\rangle) & \xrightarrow{5}_{\mathsf{n}} & (N.5) \\
(\downarrow, \langle\rangle) & \xrightarrow{20}_{\mathsf{n}} & (N.5) \\
\vdots & &
\end{array}
$$

Fig. 2.4: Example of network behaviour.

2.3.3 Process Model

bCANDLE uses a simple process-algebraic language to describe the behaviour of processes. A process is either a primitive process or a composition of other processes. There are four kinds of primitive processes in bCANDLE:

1. $k!i.x$ – *non-blocking send*: it causes the message $i.v$ to be queued instantaneously for transmission on channel k, where i is the id of the message and v is the current value of x. Then it terminates immediately.

2. $k?i.x$ – *blocking receive*: it idles until an i-message reaches its acceptance point during transmission on channel k. Then it immediately assigns the

data value of the message to the variable x and terminates.

3. $[\omega : t_1, t2]$ – *time-bounded computation*: it terminates not earlier than t_1 , and not later than t_2 , time units after beginning execution, and it atomically transforms the data state at the instant of termination as specified by the operation ω.

4. $\gamma \rightarrow P$ – *evaluate guard*: it idles until the data environment satisfies the guard γ (which is a predicate on the data state), then it performs P.

These basic processes can be compounded using a small set of operators: sequential composition, choice, interrupt, parallel composition, and process recursion, in order to construct a larger process:

- $P; Q$ (sequential composition) is a process that behaves as Q when P terminates.
- $P + Q$ (choice) is a process that can behave either like P or like Q depending on which process can first perform an action.
- $P[> Q$ (interrupt) it behaves as P until either Q performs an action or P terminates.
- $P|Q$ is the parallel composition of P and Q.
- $recX.P$ is a recursive process which has repetitive behaviour, where X is a process variable and P is a process term.

For details on the formal semantics of these process terms, one can refer to (Kendall, 2001b, p. 82).

2.3.4 Example of bCANDLE

The bCANDLE model of the flow regulator example presented in Chapter 1 is shown in Fig 2.5. It consists of two processes: *Flow* and *Valve*. *Flow* models

a process which periodically reads a flow sensor and broadcasts its value in a flow message. *Valve* models a process which repeatedly waits to receive a flow message, executes a control algorithm to calculate a new value for the valve position, and instructs an actuator to move the valve to its new position. The time bounds given to each computation represent the lower and the upper bound on the time taken to execute the computation. For example, the execution time of the software to read the flow sensor may take between 85 to 100 time units.

```
Flow | Valve
where
Flow = [ReadSensor:85,100];k!flow.x;idle
          [>
       [timer:10000,10250];Flow
Valve = k?flow.y;[AdjustValve:200,300]; Valve
network
  /*        pri  lb   ub  lB  uB  */
  k = (flow : 1, 86, 106, 24, 24)
data x, y
```

Fig. 2.5: Flow regulator in bCANDLE.

Channel `k` models a CAN bus which transmits messages of type `flow` with a priority equals 1 and transmission latency of between 86 and 106 time units from start of a transmission to an acceptance point, and a latency of between 24 and 24 time units from the acceptance point to bus idle respectively. In the following we demonstrate how transmission latency function is calculated.

The CAN protocol employs a special technique, called bit stuffing. After every 5 consecutive transmitted bits of the same value, a stuff bit is inserted of the opposite value. This ensures that there are sufficient transitions in the bit stream to ensure that the nodes can remain synchronised. The equation of (Davis et al., 2007) gives the maximum transmission time of a message C_m containing n data bytes and including stuff bits:

$$C_m = \left(g + 8.n + 13 + \left\lfloor \frac{g + 8.n - 1}{4} \right\rfloor \right) t_{bit} \tag{2.1}$$

where g is 34 for the standard format or 54 for the extended format. t_{bit} is the transmission time of a single bit. The term $\left\lfloor \frac{g + 8.n - 1}{4} \right\rfloor$ of equation 2.1 calculates the maximum number of stuff bits. The denominator of the fraction is 4 because bit stuffing includes also the stuffed bits in the frame. It is assumed for this particular example that the message acceptance point coincides with the leading edge of bit ACK0, see Fig. 1.2. In practice, the location of the acceptance point may vary from one type of CAN controller to another. In a CAN packet with n data bytes, there are $g + 8.n + 1$ bits from SOF up to, but not including, ACK0. Bit stuffing occurs from SOF up to, but not including, DEL. Therefore, the greatest number of bits is: $\left\lfloor \frac{g + 8.n - 1}{4} \right\rfloor$ which are transmitted before the acceptance point. Therefore, the lower and upper bound of time taken during the pre-acceptance phase can be calculated by equation 2.2 and 2.3.

$$\mathtt{lb} = (g + 8.n + 1)\, t_{bit} \tag{2.2}$$

$$\mathtt{ub} = \left(g + 8.n + 1 + \left\lfloor \frac{g + 8.n - 1}{4} \right\rfloor\right) t_{bit} \tag{2.3}$$

The remaining bits from ACK0 up to Int do not include stuff bits and so they equal 12 bits. Therefore, the lower and upper bound of time time taken during the post-acceptance phase can be calculated by:

$$\mathtt{lB} = \mathtt{uB} = 12.t_{bit} \tag{2.4}$$

Notice that $\mathtt{ub} + \mathtt{uB} = C_m$ is always true.

For example, in a standard CAN packet, supposing a CAN bus operating at $5 \times 10^5 bit/s$, the transmission latency function $\delta(m)$ for a message m with 1 byte size of data (i.e, $n = 1$) is:

- $\texttt{lb} = (34 + 8 + 1)\, t_{bit} = (43)5 \times 10^5 = 86\mu s$
- $\texttt{ub} = \left(34 + 8 + 1 + \left\lfloor \frac{34 + 8 - 1}{4} \right\rfloor\right) t_{bit} = (53)5 \times 10^5 = 106\mu s$
- $\texttt{lB} = \texttt{uB} = (12)5 \times 10^5 = 24\mu s$

Consequently, the transmission latency function $\delta(m) = (86, 106, 24, 24)$ for the channel k.

2.4 The Net: The intermediate model of bCANDLE

The first stage in construction of a TA model from a bCANDLE model was performed (in (Kendall, 2001b)) by translating the bCANDLE model into an intermediate net representation which is similar to a Petri net (Murata, 1989). A similar approach was applied by Garavel in the translation of LOTOS (Garavel and Sifakis, 1990), and by Yovine in the translation of ATP (Yovine, 1993). The nets which are used in bCANDLE are similar to the extended nets of (Yovine, 1993). The aim of this work is to make use of the net representation developed by the TA translator to generate an executable code. This will be discussed in details in Chapter 3. In this section, we introduce the net and present its formal semantics.

The net, as usual, consists of a set of places and a set of transitions. The net is extended in two ways. First, each transition has an associated attribute which is used to determine whether the transition is fireable or not in a given system context, where a context consists of a network and data environment. Second, a transition is associated with a set of places vulnerable to the firing of the transition. When a transition fires, control is removed not only from the places in its source set but also from all those places which are vulnerable to

it. This extension allows a compact representation of the interrupt operator in particular.

Fig. 2.6 shows the net representing the flow regulator example. Places of the net are shown as circles and transitions as boxes. The shaded circles represent a distinguished place, tick, modelling termination. A label inside a transition box refers to the transition attribute. The standard flow relation is shown using solid lines. The vulnerability relation is shown using dashed lines. A small black circle in a place shows that the place is marked. For example, transition 2 has associated attribute $[ReadSensor : 85, 100]$. The places: 2, 3 and 1 are vulnerable to the firing of transition 4. The places 2, 4 and 5 are initially marked in the net.

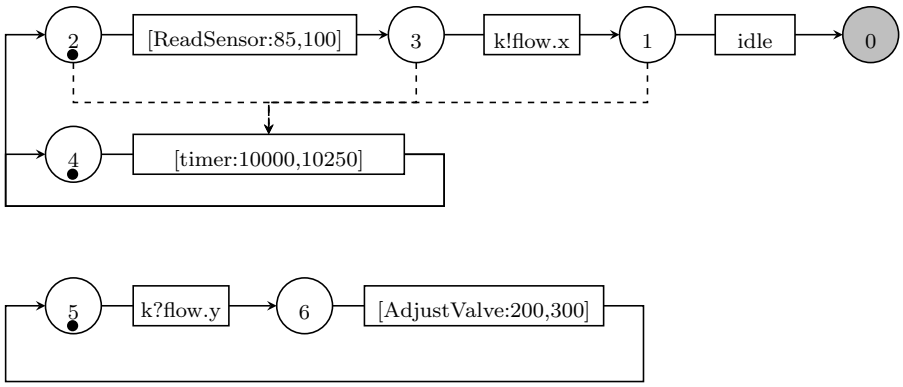


Fig. 2.6: Net of the flow regulator example.

2.4.1 Definitions and Notation

A net is defined formally as a tuple $\mathcal{R} = (W, \Theta, W^I)$ where:

- W is the set of places.
- Θ is the set of transitions.
- W^I is the set of initial marked places.

A transition $\theta = (w, W^V, \alpha, W^T) \in \Theta$ where:

- $w \in W$ is the trigger of θ, denoted ${}^{\bullet}\theta$.
- $W^V \subseteq W$ is the set of places vulnerable to θ, denoted ${}^{\circ}\theta$.
- α is the attribute of θ, denoted $\alpha\theta$.
- $W^T \subseteq W$ is the target set of θ, denoted $\theta^{\bullet}$

A place w is a trigger of exactly one transition. Transition attributes are just basic processes. The set *Attribute* is defined by the grammar:

$$\alpha ::= \widehat{\beta} | \langle \gamma \rangle | X$$

where $\alpha \in$ *Attribute* is a transition attribute. $X \in \chi$ is a process variable. $\langle \gamma \rangle$ denotes a transition attribute which consists of the predicate $\gamma \in \Gamma$. $\widehat{\beta}$ is a clocked basic process term ($k!i.x$, $k?i.x$, or $[\omega : t_1, t_2]^h$). Timed automata use clock variables to model passing the time. At the first step in the translation of bCANDLE to timed automata, explicit clock variables are introduced into the process terms and the network channels of the bCANDLE model. A computation, $[\omega : t_1, t_2]$, and its associated clock variable, h, are written as $[\omega : t_1, t_2]^h$, and a process term, P, and network channel, N, when decorated with clock variables, are written as $\widehat{P}$ and $\widehat{N}$ respectively. The set of bCANDLE systems following clock allocation is written as $\widehat{bCAN}$. The set of clock variables is represented by H and h ranges over H.

The net of the flow regulator example shown in Fig. 2.6 is then defined formally as follows:

$$\mathcal{R} = (\{0, 1, 2, 3, 4, 5, 6\}, \{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}, \{2, 4, 5\})$$

where

$$\theta_1 = \{1, \{\}, idle, \{0\}\}$$

$\theta_2 = \{2, \{\}, [ReadSensor : 85, 100], \{3\}\}$

$\theta_3 = \{3, \{\}, k!flow.x, \{1\}\}$

$\theta_4 = \{4, \{2, 3, 1\}, [timer : 10000, 10250], \{2, 4\}\}$

$\theta_5 = \{5, \{\}, k?flow.y, \{6\}\}$

$\theta_6 = \{6, \{\}, [AdjustValve : 200, 300], \{5\}\}$

2.4.2 Behaviour

The behaviour semantics of a net is given as a transition system between states consisting of a marking and a system context comprising a network and a data environment. Given a net: $\mathcal{R} = (W, \Theta, W^I)$, a system can evolve from one state $(W_1, \widehat{N}, D)$ to another state $(W_2, \widehat{N}', D')$ as result of either a process transition or a network transition where $W_1 \subseteq W$ is a marking of $\mathcal{R}$, $\widehat{N}$ is network context, and D is data context. In a process transition: for a trigger $w \in W$ of a transition θ, if the context of $\widehat{N}$ and D satisfies conditions required by the attribute α of θ, then new marking W_2 is created from W_1 by removing w and vulnerable places to θ, and then including target places of θ. The new context, $\widehat{N}'$, D' is created according to the requirements of the attribute. In the case of a network transition, the system can evolve to a new state if a network component is modified, but marking and data environment remain unchanged. The process transitions of $(W_1, \widehat{N}, D)$ are given by the rule **R.1**, and the network transitions by the rule **R.2**.

$$\textbf{R.1}\quad \frac{\begin{array}{c} w \in W_1 \wedge (w, W^V, \alpha, W^T) \in \Theta \wedge \mathsf{fire}(\alpha, \widehat{N}, D, \psi, \lambda, \mathsf{H}', \widehat{N}', D') \wedge \\ W_2 = W_1 \setminus (w \cup W^V) \cup W^T \wedge \mathsf{H} = \mathsf{H}' \cup \mathsf{clk}(W^T) \end{array}}{(W_1, \widehat{N}, D) \xrightarrow{\psi, \lambda, \mathsf{H}}_{\mathcal{R}} (W_2, \widehat{N}', D')}$$

$$\textbf{R.2}\ \frac{\begin{array}{c}\widehat{N} \xrightarrow{\psi,\lambda_n,\mathsf{H}}_{\mathsf{n}} \widehat{N}' \wedge \\ \forall k \in K, i \in I.(\neg \mathsf{awaited}(W,k,i) \vee \widehat{N}_k \neq (\uparrow i._-, \_) \vee \widehat{N}_k = \widehat{N}'_k)\end{array}}{(W,\widehat{N},D) \xrightarrow{\psi,\lambda_n,\mathsf{H}}_{\mathcal{R}} (W,\widehat{N}',D)}$$

The fire relation, as given in Fig. 2.7, simply determines the semantic rules for the basic processes. λ is an action label, λ_n is a network action label, H is a set of clocks, ψ is a clock constraint, K is a set of channel identifiers, and I is a set of message identifiers. h_u is a distinct clock variable used to model an urgent transition (executed without delay). $\mathsf{clk}(W^T)$ denotes a set of clocks associated with the set W^T. $\mathsf{awaited}(W,k,i)$ holds iff, in the marking W, it is possible to receive from channel k a message with identifier i. Formally,

$$\mathsf{awaited}(W,k,i) \widehat{=} \{w \in W | \alpha\theta_w = k?i._-\} \neq \emptyset$$

$$\textbf{F\_Snd}\ \frac{\widehat{N}_k = (s,u) \wedge v = D.x}{\mathsf{fire}(k!i.x, \widehat{N}, D, \mathsf{tt}, k!i.v, \{h_u\}, \widehat{N}[k := (s, u \nleftarrow i.v)], D)}$$

$$\textbf{F\_Rcv}\ \frac{\widehat{N}_k = (\uparrow i.v, \_)}{\mathsf{fire}(k?i.x, \widehat{N}, D, \mathsf{tt}, k?i.v, \{h_u\}, \widehat{N}, D[x := v])}$$

$$\textbf{F\_Comp}\ \frac{D \xrightarrow{\omega}_d D' \wedge t_1 \in \mathbb{N}}{\mathsf{fire}([\omega : t_1, t_2]^h, \widehat{N}, D, h \geq t_1, \omega, \{h_u\}, \widehat{N}, D')}$$

$$\textbf{F\_Gu}\ \frac{D \models \gamma}{\mathsf{fire}(\langle\gamma\rangle, \widehat{N}, D, \mathsf{tt}, \gamma, \{h_u\}, \widehat{N}, D)}$$

Fig. 2.7: Rules for fire.

We are interested only in the process transition when generating code because both marking of the net and the data state can be changed. The network transition can only modify the network state and we assume that the behaviour defined by this rule is handled by the CAN controller. The following example shows an example of a possible behaviour of a net.

2.4.3 Example of Net Behaviour

In this example, we give a possible behaviour of the net of the flow regulator example shown in Fig. 2.6. The following conventions are adopted:

- A system state is shown as a tuple $(W, \widehat{N}, D)$.
- D is the initial data state where all variables are set to zero. $D[var := val]$ denotes a data state D' which is the same as D except that the variable var is associated with the value val in D'.
- The network component $\widehat{N}$ shows only the dynamic attributes of the single channel k.
- Time delay is chosen arbitrarily from the allowable range of values.
- The transmitted values of the actual flow sensor reading are abstracted, where 0 represents a reading in the low level, and 1 represents a reading in the high level.

A possible behaviour of the net is illustrated in Fig. 2.8. The net trace starts from the initial state $(\{2, 4, 5\}, (\downarrow, \langle\rangle), D)$ and evolves to a new state as result of either the process transition rule **R.1** or the network transition rule **R.2**.

2.5 The Translation of bCANDLE to Net

For a bCANDLE system $(\widehat{P}, \widehat{N}, D) \in \widehat{bCAN}$, we briefly give in the following the translation rules of constructing the net for $\widehat{P}$, denoted $\mathcal{N}[\![\widehat{P}]\!]$. We respectively consider the basic terms, the guard $\gamma \rightarrow \widehat{P}$, the compound terms $\widehat{P}_1; \widehat{P}_2$, $\widehat{P}_1 + \widehat{P}_2$, $\widehat{P}_1 [> \widehat{P}_2$, and $\widehat{P}_1 | \widehat{P}_2$, the process variable, and the recursion operator. For more details about the net construction from a bCANDLE system, one can refer to (Kendall, 2001b, p. 112).

$$
\begin{array}{lll}
(\{2,4,5\},(\downarrow,\langle\rangle),D) & \xrightarrow{100} & \text{R.2} \\
(\{2,4,5\},(\downarrow,\langle\rangle),D) & \xrightarrow{ReadSensor} & \text{R.1} \\
(\{3,4,5\},(\downarrow,\langle\rangle),D[x:=1]) & \xrightarrow{k!flow.1} & \text{R.1} \\
(\{1,4,5\},(\downarrow,\langle flow.1\rangle),D[x:=1]) & \xrightarrow{k\rightsquigarrow flow.1} & \text{R.2} \\
(\{1,4,5\},(\overset{86,106}{\rightsquigarrow} flow.1,\langle\rangle),D[x:=1]) & \xrightarrow{100} & \text{R.2} \\
(\{1,4,5\},(\overset{0,6}{\rightsquigarrow} flow.1,\langle\rangle),D[x:=1]) & \xrightarrow{k\uparrow flow.1} & \text{R.2} \\
(\{1,4,5\},(\uparrow flow.1,\langle\rangle),D[x:=1]) & \xrightarrow{k?flow.1} & \text{R.1} \\
(\{1,4,6\},(\uparrow flow.1,\langle\rangle),D[x:=1,y:=1]) & \xrightarrow{flow.1\rightsquigarrow k} & \text{R.2} \\
(\{1,4,6\},(flow.1 \overset{24,24}{\rightsquigarrow},\langle\rangle),D[x:=1,y:=1]) & \xrightarrow{24} & \text{R.2} \\
(\{1,4,6\},(flow.1 \overset{0,0}{\rightsquigarrow},\langle\rangle),D[x:=1,y:=1]) & \xrightarrow{k\downarrow} & \text{R.2} \\
(\{1,4,6\},(\downarrow,\langle\rangle),D[x:=1,y:=1]) & \xrightarrow{200} & \text{R.2} \\
(\{1,4,6\},(\downarrow,\langle\rangle),D[x:=1,y:=1]) & \xrightarrow{AdjustValve} & \text{R.1} \\
(\{1,4,5\},(\downarrow,\langle\rangle),D[x:=1,y:=1]) & \xrightarrow{10000} & \text{R.2} \\
(\{1,4,5\},(\downarrow,\langle\rangle),D[x:=1,y:=1]) & \xrightarrow{timer} & \text{R.1} \\
(\{2,4,5\},(\downarrow,\langle\rangle),D[x:=1,y:=1]) & \longrightarrow & \\
\vdots & &
\end{array}
$$

Fig. 2.8: Example of net behaviour.

Basic terms: Let $\widehat{\beta}$ be one of the clocked basic terms $k!i.x$, $k?i.x$ or $[\omega : t_1, t_2]^h$, then the net of $\widehat{\beta}$ is constructed as follows:

$$\mathcal{N}[\![\widehat{\beta}]\!] \mathrel{\widehat{=}} (\{w\}, \{(w, \{\}, \widehat{\beta}, \{\mathsf{tick}\})\}, \{w\})$$

Guard: Let $\mathcal{N}[\![\widehat{P}]\!] = (W, \Theta, W^I)$, then the net of $\gamma \to \widehat{P}$ is given by

$$\mathcal{N}[\![\gamma \to \widehat{P}]\!] \mathrel{\widehat{=}} (W \cup \{w\}, \Theta \cup \{(w, \{\}, \langle\gamma\rangle, W^I)\}, \{w\})$$

Sequential composition: Let $(W_i, \Theta_i, W_i^I) = \mathcal{N}[\![\widehat{P}_i]\!]$, for $i \in \{1, 2\}$, be disjoint nets. The net $\mathcal{N}[\![\widehat{P}_1; \widehat{P}_2]\!]$ for the sequential composition $\widehat{P}_1; \widehat{P}_2$ is given by

$$\mathcal{N}[\![\widehat{P}_1; \widehat{P}_2]\!] \mathrel{\widehat{=}} (W_1 \cup W_2, \Theta_1' \cup \Theta_2, W_1^I)$$

where

$$\Theta_1^{'} \mathrel{\widehat{=}} \{\theta \mid \theta \in \Theta_1 \wedge \theta^\bullet \neq \{\mathsf{tick}\}\}$$
$$\cup \{({}^\bullet\theta, {}^\circ\theta, \alpha\theta, W_2^I) \mid \theta \in \Theta_1 \wedge \theta^\bullet = \{\mathsf{tick}\}\}$$

Choice: Let $(W_i, \Theta_i, W_i^I) = \mathcal{N}[\![\widehat{P}_i]\!]$, for $i \in \{1, 2\}$, be disjoint nets. Then

$$\mathcal{N}[\![\widehat{P}_1 + \widehat{P}_2]\!] \mathrel{\widehat{=}} (W_1 \cup W_2, \Theta, W_1^I \cup W_2^I)$$

where

$$\Theta \mathrel{\widehat{=}} \{\theta \mid \theta \in \Theta_1 \wedge^\bullet \theta \notin W_1^I\}$$
$$\cup \{({}^\bullet\theta, {}^\circ\theta \cup W_2^I, \alpha\theta, \theta^\bullet) \mid \theta \in \Theta_1 \wedge^\bullet \theta \in W_1^I\}$$
$$\cup \{\theta \mid \theta \in \Theta_2 \wedge^\bullet \theta \notin W_2^I\}$$
$$\cup \{({}^\bullet\theta, {}^\circ\theta \cup W_1^I, \alpha\theta, \theta^\bullet) \mid \theta \in \Theta_2 \wedge^\bullet \theta \in W_2^I\}$$

Interrupt: Let $(W_i, \Theta_i, W_i^I) = \mathcal{N}[\![\widehat{P}_i]\!]$, for $i \in \{1, 2\}$, be disjoint nets. Then

$$\mathcal{N}[\![\widehat{P}_1 [> \widehat{P}_2]\!] \mathrel{\widehat{=}} (W_1 \cup W_2, \Theta, W_1^I \cup W_2^I)$$

where

$$\Theta \mathrel{\widehat{=}} \{\theta \mid \theta \in \Theta_1 \wedge \theta^\bullet \neq \{tick\}\}$$
$$\cup \{({}^\bullet\theta, {}^\circ\theta \cup W_2^I, \alpha\theta, \theta^\bullet) \mid \theta \in \Theta_1 \wedge \theta^\bullet = \{tick\}\}$$
$$\cup \{\theta \mid \theta \in \Theta_2 \wedge^\bullet \theta \notin W_2^I\}$$
$$\cup \{({}^\bullet\theta, {}^\circ\theta \cup W_1, \alpha\theta, \theta^\bullet) \mid \theta \in \Theta_2 \wedge^\bullet \theta \in W_2^I\}$$

Parallel composition: Let $(W_i, \Theta_i, W_i^I) = \mathcal{N}[\![\widehat{P}_i]\!]$, for $i \in \{1, 2\}$, be disjoint nets. Then

$$\mathcal{N}[\![\widehat{P}_1 | \widehat{P}_2]\!] \mathrel{\widehat{=}} (W_1 \cup W_2, \Theta_1 \cup \Theta_2, W_1^I \cup W_2^I)$$

Process variable: Let X be a process variable, then the net of X is defined:

$$\mathcal{N}[\![X]\!] \widehat{=} (\{w\}, \{(w, \{\}, X, \{\})\}), \{w\})$$

where $w \neq \mathsf{tick}$ is a place.

Recursion operator: Let $\mathcal{N}[\![\widehat{P}]\!] = (W, \Theta, W^I)$, then the net of $rec\ X.\widehat{P}$ is given by

$$\mathcal{N}[\![rec\ X.\widehat{P}]\!] \widehat{=} (W, \Theta', W^I)$$

where

$$\Theta' \widehat{=} \{\theta \mid \theta \in \Theta \wedge \alpha\theta \neq X\}$$
$$\cup \{({}^\bullet\theta, {}^\circ\theta, \alpha\theta, W^I) \mid \theta \in \Theta \wedge \alpha\theta = X\}$$

2.6 The CANDLE Programming Language

bCANDLE is very low-level for system developers to be used to write a system description. Therefore, CANDLE (Kendall, 2001b) was introduced for the purpose of a system design. CANDLE (Kendall et al., 1998a; Kendall, 2001a,b) is a high-level programming language intended for distributed embedded systems based on the CAN network. The formal semantics of the language is defined by translation into bCANDLE. A CANDLE program consists of a number of *processes* which implement a system behaviour.

Fig 2.9 shows two CANDLE processes representing the flow regulator example. The CANDLE which is used in this work is a simplified version of the original one. More elaborated language is available in (Kendall, 2001b). The first process implements the behaviour of the flow task. The second process implements the behaviour of the valve task. The actual definition of the data variables (`fFlow` and `vFlow`) and the operations (`readSensor` and `adjustValve`) are assumed to be provided from an external language (e.g., C). The two pro-

```
Flow | Valve

where

Flow =
  every 10000 do
    readSensor();
    snd(k, FLOW, fFlow)
  end every

Valve =
  loop do
    rcv(k, FLOW, vFlow);
    adjustValve()
  end loop
```

Fig. 2.9: Flow regulator in CANDLE.

cesses are composed together using the parallel operator (|). In addition to primitive statements (e.g., `snd`, `rcv`, and procedure calls), the language features constructs to control flow of a program, like any traditional programming language, such as branching, loops, and exceptions handling. In the following, we informally introduce the compound statements of the language.

Compound Statements

Case Statement

In the `case` statement, one of several statements is executed depending on the evaluated value of an expression. The statement has the following general form:

```
case e
   when v0 => s0
   when v1 => s1
   ⋮
   when vn => sn
```

```
    otherwise => s
  end case
```

When the value of the expression e is evaluated, one of the corresponding statement s_i will be executed when v_i matches the return value of the expression. If none of them are matched, the statement s is executed.

Select statement

The statement allows a choice between a number of statements to be executed depending on the reception of a message or the elapse of a time. The `select` statement has the following form:

```
select
  when rcv(k_1, i_1.x_1) => s_1
  when rcv(k_2, i_2.x_2) => s_2
  ⋮
  when rcv(k_n, i_n.x_n) => s_n
  when timeout elapse(t) => s
end select
```

If the message i_j is successfully received, then the statement s_j is executed. If more than one message is received, then the choice between which statement is executed, is made non-deterministically. If no message is received before t time units, then the statement s is executed. Additionally, CANDLE provides an extended form of the statement as follows:

```
select
  when rcv(k_1, i_1.x_1) => s_1
  when rcv(k_2, i_2.x_2) => s_2
  ⋮
```

```
    when rcv(k_n, i_n.x_n) => s_n
    when elapse(t) => s
  in
    body
  end select
```

The statement behaves similarly to the original one except that the body statement is executing while waiting the message reception and the time expiry. When a message is received or a time delay is expired, then the execution of *body* is interrupted, and the execution of the corresponding statement is started.

Loop statement

The `loop` statement allows an expression to be executed repeatedly forever. The statement has the following form:

```
loop do
   s
end loop
```

where s is a statement.

Every statement

The `every` statement allows an expression to be executed periodically. The statement has the following form:

```
every T do
   s
end every
```

where s is a statement that runs periodically every T time units. The body of this statement is initiated immediately.

Trap and Exit statements

The `trap` is used to handle exceptions produced in a program block. The statement has the following general form:

```
trap
   when x1 => s1
   when x2 => s2
   ⋮
   when xn => sn
in
  body
end trap
```

where x_i is an exception identifier and s_i is a statement that acts as a handler for the exception. The body of the `trap` begins executing. An exception can be raised inside *body* using the `exit` statement, e.g.

```
exit xi
```

If an exception is trapped, the execution of *body* is interrupted and the associate exception handler begins executing.

2.7 The Translation of CANDLE to bCANDLE

In this section, we briefly give the rules of translating the CANDLE statements into bCANDLE. We consider the primitive statements first and the compound statements after that. For more details about the construction of a bCANDLE model from a CANDLE program, one can refer to (Kendall, 2001b, p. 164).

Null and Idle statements

$[\![\texttt{null}]\!] \mathrel{\widehat{=}} \texttt{null}$

$[\![\texttt{idle}]\!] \mathrel{\widehat{=}} \texttt{idle}$

Send and Receive statements

$[\![\texttt{snd}(k, i.e)]\!] \mathrel{\widehat{=}} k!i.x$

$[\![\texttt{rcv}(k, i.e)]\!] \mathrel{\widehat{=}} k?i.x;$

Elapse statement

$[\![\texttt{elapse}(T)]\!] \mathrel{\widehat{=}} [timer: \ T]$

Procedure Call

$[\![P(e_1, ..., e_n)]\!] \mathrel{\widehat{=}} [\omega: \ t^{lb}, t^{ub}]$

where P is the name of the procedure, e_i is an expression that expresses a parameter of P, and t^{lb} (resp. t^{ub}) is the lower bound (resp. upper bound) on the time required to execute the procedure and to evaluate its parameters.

Case statement

$[\![\texttt{case } e \texttt{ when } v_0 \texttt{=>} s_0 \texttt{ when } v_1 \texttt{=>} s_1 \ldots \texttt{when } v_n \texttt{=>} s_n \texttt{ otherwise=>} s \texttt{ end case}]\!] \mathrel{\widehat{=}}$
$[t^{lb}, t^{ub}]; (\gamma_0 \rightarrow [\![s_0]\!] \ + \ \gamma_1 \rightarrow [\![s_1]\!] \ + \ldots \ + \gamma_n \rightarrow [\![s_n]\!] + \gamma \rightarrow [\![s]\!])$

where t^{lb} (resp. t^{ub}) denotes the lower bound (resp. upper bound) on the time required to complete the evaluation of e, γ_i is *true* iff the expression e equals v_i, and the guard γ is *true* when iff $\neg(\gamma_0 \vee \gamma_1 \ldots \vee \gamma_n)$.

Select statement

$[\![\texttt{select when } g_0 \texttt{=>} s_1 \texttt{ when } g_1 \texttt{=>} s_1 \ldots \texttt{when } g_n \texttt{=>} s_n \texttt{ end select}]\!] \mathrel{\widehat{=}}$
$(\beta_0; s_0 + \beta_1; s_1 + \ldots + \beta_n; s_n)$

where $[\![g]\!] = \beta$, and β is either $k?i.x$ or $[timer : t]$.

The extended `select` statement is translated into bCANDLE as follows:

$$[\![\texttt{select when } g_0 \texttt{=>} s_0 \texttt{ when } g_1 \texttt{=>} s_1 \ldots \texttt{when } g_n \texttt{=>} s_n \texttt{ in } s \texttt{ end select }]\!] \mathrel{\widehat{=}} ([\![s]\!] \ [> \ (\beta_0; s_0 + \beta_1; s_1 + \ldots + \beta_n; s_n))$$

Loop statement

$$[\![\texttt{loop do } s \texttt{ end loop}]\!] \mathrel{\widehat{=}} \texttt{rec } LOOP.[\![s]\!]; LOOP$$

where $LOOP$ is a new process variable.

Every statement

$$[\![\texttt{every } T \texttt{ do } s \texttt{ end every}]\!] \mathrel{\widehat{=}} [\![\texttt{loop do select when elapse T in s;idle end select end loop}]\!]$$

Trap and Exit statements

$$[\![\texttt{trap when } x_0 \texttt{=>} s_0 \texttt{ when } x_1 \texttt{=>} s_1 \ldots \texttt{when } x_n \texttt{=>} s_n \texttt{ in } s \texttt{ end trap}]\!] \mathrel{\widehat{=}} ([\![s]\!][>(\gamma_0 \rightarrow [\![s_0]\!] + \gamma_1 \rightarrow [\![s_1]\!] + \ldots + \gamma_n \rightarrow [\![s_n]\!]))$$

where the guard γ_i is the *true* when the exception x_i is raised.

$$[\![\texttt{exit } x]\!] \mathrel{\widehat{=}} [exit\_x : t^{lb}, t^{ub}]; idle$$

where $exit\_x$ is an operation required to raise the exception x.

Sequential and Parallel Composition

$$[\![s_1 \ ; \ s_2]\!] \mathrel{\widehat{=}} [\![s_1]\!] \ ; \ [\![s_2]\!]$$
$$[\![s_1 \mid s_2]\!] \mathrel{\widehat{=}} [\![s_1]\!] \mid [\![s_2]\!]$$

2.8 Summary

In this chapter, an overview of the code and model generation approach has been provided. The main assumptions of the target distributed embedded systems have been presented. The essential component of the approach which is the bCANDLE modelling language has been introduced. The net, the intermediate representation model at which our code and model generator are based, has been defined. Because bCANDLE is very-low level for system developers, the CANDLE high-level language is introduced. The translation rules of CANDLE to bCANDLE and of bCANDLE to net have been outlined.

3. CANDLE CODE GENERATOR

3.1 Introduction

We aim in this chapter to demonstrate how an implementation is generated from CANDLE. Our approach generates executable C code from the intermediate net representation. The C language is chosen because is widely supported and has available cross-compilers and static analysis tools. A program written in C is not tied to a particular hardware platform and so it can be easily ported. There are particular reasons for using the net to derive the implementation. First, the net is a simple and compact representation which can yield a small size of code suitable to a resource-constrained embedded system. Second, there are some CANDLE constructs that lead to similar nets at translation. Therefore, implementing the net can reduce the effort required to generate code for each CANDLE statement. Third, the behaviour of the implementation should be designed in a way that matches the semantics of CANDLE. This problem is now scaled down to the problem of generating code which only needs to respect the semantics of the net.

The chapter is organised as follows. Section 3.2 introduces our implementation model to execute the net. Section 3.3 presents the representation of the net in the C language. The implementation of CANDLE channels is explained in section 3.4. A simple architecture language is presented in section 3.5. The language is adopted to describe the system architecture including hardware and software components. Finally, section 3.6 concludes the chapter.

3.2 The Implementation Model

This section defines the implementation model to execute a net. This model has been adopted in order to restrict the run-time behaviour of the system in a way that allows off-line prediction for the execution time bounds of system components. The system then can be analysed to verify that it conforms to a specification and that sufficient run-time resources are available.

3.2.1 Features and Notation

Fig. 3.1 shows some features of the implementation model and introduces notation used in its description. The implementation is clock-driven: a timer provides a single interrupt source and a periodic interrupt to the implementation. The period of the interrupt is denoted by T. The interrupt is serviced by an interrupt service routine (ISR) that takes Cs time units to execute on each invocation. The time Cs comprises the maximum time to execute the ISR algorithm, presented in section 3.3.2, and the maximum time to enter and leave the ISR. If the actual execution time of the ISR is less than Cs for any invocation then the completion of the ISR is deferred until Cs time units have elapsed. This helps to reduce jitter in the system. The ISR is assumed to contain a notional *reaction instant*, ρ, at which point all executable instantaneous actions are deemed to be performed. It is not known when this instant will happen exactly inside the ISR, but it could be at any time between the beginning and the end of the ISR. This non-determinism in the model allows the omission of many details in the model of the ISR, simplifying it and making model checking more tractable.

A reaction may include the following 'instantaneous' actions:

- update active soft timers,
- transfer external CAN messages to intermediate ISR buffers,

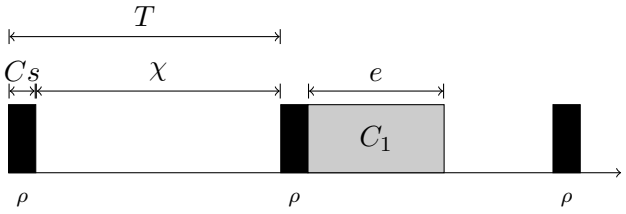


Fig. 3.1: Features and notation of the implementation model.

- identify the termination of computations and fire associated transitions,
- transfer data to transmit buffers for any active $k!i.x$ transitions and fire the transitions,
- transfer message data, if available, from intermediate ISR buffers to process variables for active $k?i.x$ transitions and fire the transitions,
- perform local communications, if available, and fire transitions,
- evaluate active guards and fire the transitions of those that are satisfied,
- release the next computation.

Fig. 3.1 illustrates the release of a computation C_1 in the reaction of the second invocation of the ISR. Its actual execution time is denoted e. The time available for process computation in any tick period is called a *computation interval* and is denoted by χ. Notice that in any tick period, a node is executing the ISR, executing some process computation or idling, i.e. $T = Cs + \chi$.

The period of the interrupt T and the ISR time Cs are not fixed; they may vary from one computing node to another. The value of T is assumed to be determined by the user at design time. Different values of T may have different effects on the real system. For example, if T is selected to be short, then the system becomes more responsive because events are noticed more quickly, but more overhead occurs because the ISR is executed more often. On the other

hand, when T is chosen to be long, the system will suffer less overhead, but it becomes less responsive.

The proposed model has been influenced by other approaches, mainly the time-triggered approach (Kopetz, 1997) and synchronous approach (Benveniste and Berry, 1991). In the following, a comparison between our approach and other approaches are discussed.

3.2.2 Comparison with Other Models

There are two distinctly different approaches to the design of real-time embedded systems: event-triggered (ET) approach and time-triggered (TT) approach. It is not the purpose of the thesis to compare the event-triggered and time-triggered approaches. Many comparisons between the two approaches have been published elsewhere, for example (Kopetz, 1991; Alber, 2004; Scarlett and Brennan, 2006; Armengaud et al., 2009). The aim here is to justify the decision behind adopting a time-triggered design to execute the net.

The majority of software architectures follow the ET paradigm. In the ET approach, all system activities (computations and communications) are initiated whenever a significant change in the environment (event) occurs, such as a time tick, a button press, or arrival of a message. The ET approach is characterised by flexibility and imposes fewer design constraints compared to the TT approach. It requires fewer assumptions such as in constructing the system architecture (Armengaud et al., 2009). Adding additional components to the system does not require changes in the other system components. However, the ET architecture uses the notation of an interrupt to observe the occurrence of the environment events. Each event can be associated with an interrupt that forces the system to react to the event by executing an appropriate computation (*task*). Because it is not known pre-runtime when the interrupts will happen, it becomes difficult to calculate the worst-case execution times of the

systems components off-line which are required to construct a model of the system. Therefore, unless events are periodic or sporadic, the temporal behaviour of the ET system becomes difficult to predict. Although there are some techniques such as *stopwatches* (Cassez and Larsen, 2000) that does not require pre-runtime calculation of worst-case execution times, using stopwatches may make some verifications undecidable (Brihaye et al., 2006; Cassez and Larsen, 2000; Henzinger et al., 1995). Additionally, the architecture requires special mechanisms (e.g., semaphore and mutex) to resolve data access sharing, and inter-task communication and synchronisation. These mechanisms are resource demanding and can make timing analysis of the system behaviour even more complex.

In the TT approach, all system activities are initiated at pre-defined points in time (Kopetz, 1997). The approach imposes a restriction on using interrupts in order to preserve the predictability of the system behaviour. There is usually one source of interrupt which is the tick timer (Kopetz, 1995). The system periodically observes the state of the environment and triggers an appropriate action according to a predefined plan. TT architecture employs a static or predefined scheduler in which the schedule of all software components is computed off-line (Xu and Parnas, 2000). Since the main characteristics of the components (periods, worst-case computation times, and deadlines) are known in advance, it is possible to verify that all timing constraints will be satisfied. Using such a scheduler strongly facilitates timing analysis of the system (Ebner, 1998). Moreover, the static scheduler provides pre-run time resolving of the timing and data dependencies and so avoiding using resource demanding mechanisms such as semaphores. However, building the TT schedule requires a number of parameters of the system components to be determined during the design time, such as the tick interval and the task order and offset. The problem is that determining these parameters is not trivial and characterised as NP-hard (Gendy and Pont, 2008). Moreover, slight changes to these parameters may lead to a

significant change in the reconstruction of the schedule. This problem is called the "fragility" of the TT design (Pont, 2008a). In contrast, our approach does not impose restricted assumptions (e.g, periodicity) on the system components. Once the bounds on the execution times are calculated, they are exported to the model generator. An abstract model is produced from the system description. Then the system behaviour can be analysed using a model-checking tool.

Additionally, the proposed approach has been influenced by the synchronous approach (Benveniste and Berry, 1991). The synchronous approach adopts a very conservative assumption called the *synchrony hypothesis* about the system behaviour. It assumes that computations and communications of the system components take zero time to execute. The system reacts to its environment instantaneously. At each reaction, it reads its inputs, performs computation, and then generates its outputs. The system components communicate through broadcast channels. During one reaction the transmitted data becomes available instantly to each receiver component. The broadcast communication mechanism behaves similarly to wires in a synchronous digital circuit (Benveniste et al., 2003). With the fully synchronous approach it is not always feasible to build systems because it is often that systems are implemented on a distributed architecture in which a set of computing nodes communicate by asynchronous means of communication. For that reason, the globally asynchronous locally synchronous (GALS) approach (Berry et al., 1993) has been proposed to unify the capabilities of the synchronous and asynchronous approaches. The system can be seen in this context as a set of synchronous components that are connected by asynchronous communication channels. In contrast, our approach adopts more flexible and realistic assumptions about the system behaviour than the the synchronous approach. Computations take time to execute. The execution times of the computations can be bounded by adopting a restrictive software architecture similar to TT architecture. The system components communicate through broadcast channels which abstract the CAN protocol. If the

components are allocated in the same computing node, they communicate via local channels in which the transmission time is assumed to occur instantly. The concept of local channels is discussed in section 3.4.2. When the components are distributed, the transmission time is not instant. Moreover, the GALS approach employs two different semantics models to express the system behaviour, which is locally synchronous and deterministic, and globally asynchronous and non-deterministic. This however makes the task of checking whether the implementation respects the semantics of its model so hard. In contrast, our approach employs a single semantics model, which is fully asynchronous and non-deterministic.

3.2.3 Scheduling

Many nodes in a CAN-based system may require only an elementary software architecture in which a single process repeatedly performs a simple function, e.g. a sensor node that samples its environment, normalises the reading given by its ADC and broadcasts the result on the CAN bus. However, some nodes may require a more complicated architecture in which the CPU is shared by computations released by multiple processes. A key requirement for the application of our method is the capability to perform offline calculation of best-case and worst-case bounds on the total time required to complete all computations, including time when a computation is ready to run but is not allocated to the CPU. The flexibility of the computation model that we allow means that, in general, it is not possible to adopt typical scheduling strategies such as fixed-priority pre-emptive or earliest-deadline-first in our implementations, since the periodicity requirements for the application of response-time analysis techniques applicable to these strategies may not be satisfied. Instead, we consider other standard strategies including round-robin and cooperative scheduling (Liu, 2000; Pont, 2008b) in which an offline analysis is possible.

These standard strategies are reviewed in the reminder of this section.

1. **Pure round-robin scheduling** is a simple scheduling approach in which the processes are ordered and each process is allocated a single computation interval ('slot') in which to execute (part of) a computation. The next slot is allocated to the next process in the ordering and so on, until all processes have had a chance to use a slot. This completes a *round* of the schedule. If there are n processes in the round-robin set, then every round is of length n. The schedule is simply the repetition of rounds executed in this fashion. This approach is reasonable for long running computations (i.e. computations that needs more than one computation interval to complete). However, if the set of processes allocated to the same node contains short computations (i.e. computations that can complete in a single computation interval), then the round-robin approach could be inefficient. For example, Fig. 3.2 shows three short computations ($C1$, $C2$, and $C3$) scheduled using the simple round-robin approach. The computations become ready to run in the first tick. Because they are dispatched in order, $C3$ requires three ticks (one round) to complete, even though it is a short computation.

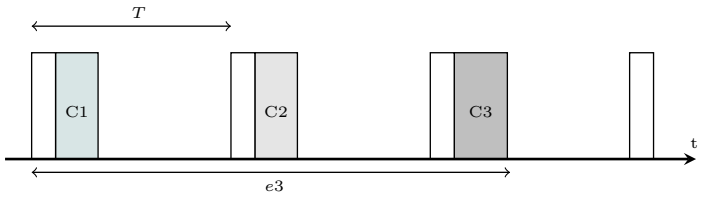


Fig. 3.2: Simple round-robin scheduling of short computations.

2. **Cooperative scheduling** may be adopted when the set of processes allocated to a node contains computations that are short enough to fit together in a single computation interval. They can be scheduled cooperatively, i.e. in any computation interval, each process with a ready computation runs to completion and then relinquishes the CPU to allow

execution of the next ready computation, if there is one (Pont, 2008b). When all ready computations have been completed, the idle process runs until the next timer interrupts. Fig 3.3 shows three short computations scheduled cooperatively. Now, because $C3$ has a chance to run in the same computation interval as $C1$ and $C2$, the computation will complete in one tick time.

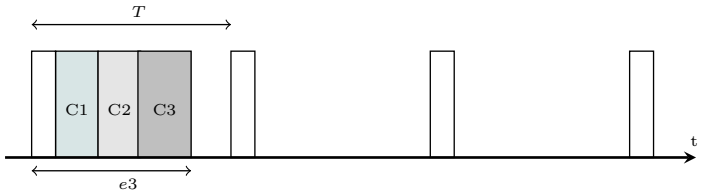


Fig. 3.3: Cooperative scheduling of short computations.

3. **Weighted round-robin scheduling** is a modification of the simple round-robin approach in which each process is assigned a *weight*. The weight of a process determines the number of consecutive slots that are allocated to it in every round. If the round-robin set comprises processes $P_1, \ldots, P_n$ and each process P_i is assigned a weight s_i then the number of slots in every round is $\sum_{i=1}^{n} s_i$. For example, Fig. 3.4 and Fig. 3.5 show two computations $C1$ and $C2$ scheduled using the simple and the weighted round-robin approach, respectively. By using the simple round-robin method, although $C1$ completes within one tick time, $C2$ will require 4 ticks (two rounds) to complete, see Fig. 3.4. However, in the modified round-robin method, we could assign $C1$ weight 1 and $C2$ weight 2, then the computation $C2$ will complete within 3 ticks (one round), see Fig. 3.5.

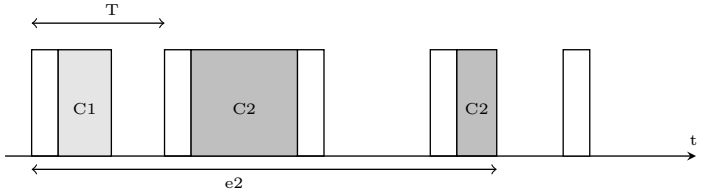


Fig. 3.4: Pure round-robin scheduling.

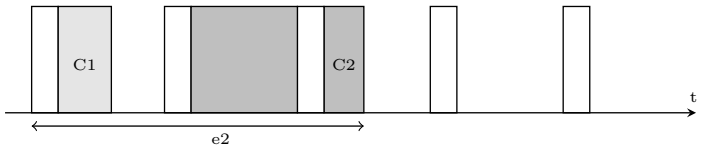


Fig. 3.5: Weighted round-robin scheduling.

4. **Hybrid scheduling** A hybrid scheduling algorithm combines cooperative scheduling and weighted round-robin scheduling. The set of processes, P, is divided into two disjoint subsets, P_{co} and P_{rr}, of cooperatively-scheduled and round-robin processes, respectively. The available computation time χ in a tick period is divided into a time interval, χ_{co}, for cooperatively-scheduled processes and a time interval, χ_{rr}, for round-robin processes. The processes in P_{co} are scheduled using cooperative scheduling during χ_{co} and the processes in P_{rr} are scheduled using weighted round-robin scheduling during χ_{rr}. The actual values for χ_{co} and χ_{rr} can be chosen freely by the system developer, as long as $\chi_{co} + \chi_{rr} = \chi$. Hybrid scheduling subsumes both cooperative and round-robin scheduling: if $P_{co} = P$ (and $\chi_{co} = \chi$) then we have pure cooperative scheduling; and if $P_{rr} = P$ (and $\chi_{rr} = \chi$) then we have pure weighted round-robin scheduling. Fig.3.6 shows an example of the hybrid scheduling of three computations. The computation $C1$ and $C2$ are scheduled cooperatively, whereas the computation $C3$ is scheduled using the weighted round robin.

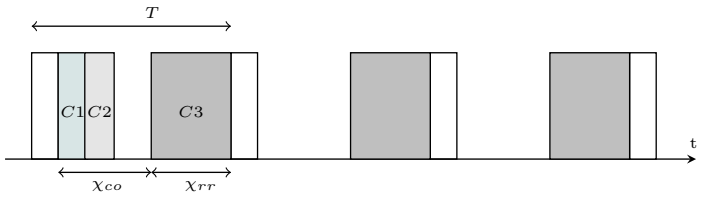


Fig. 3.6: Example of hybrid scheduling.

Four static scheduling approaches have been proposed. For the purpose of this study, we implemented only the pure round-robin and the cooperative scheduler. The choice between using one of them is performed offline by the system

developer. For example, if a process has a computation that updates a slow device, then it should not be scheduled cooperatively with other processes because this kind of computation is considerably slower than that which illuminates a LED for example. A general analysis of the response times of computations under these scheduling policies is presented in section 4.2.4. The implementation of the other methods will be considered in future work.

3.3 Representation of the Net

Embedded systems have constrained resources (CPU and memory) so an efficient executable code in terms of space and performance is a common requirement. In this section, we demonstrate how a net is implemented in C. Efficient data structures are proposed to store a net in memory. Attention has been paid to splitting the code between read-only memory (ROM) and random-access memory (RAM). Storing part of the code in ROM prevents unexpected changes. For example, a stack may overwrite the program data as it shares RAM area with the program. Moreover, minimising the required area of RAM reduces the total cost of the embedded system because RAM is more expensive than ROM storage chips. In the following, the data structures used to represent the net are presented in section 3.3.1. The pseudo code the ISR that updates the net state is presented in section 3.3.2.

3.3.1 Data Representation

The main architecture of the net is described using a UML class diagram depicted in Fig. 3.7. In this figure, the class Net consists of *places*, *marked places* and a number of transitions. A transition is represented by the class $Transition$ which consists of *trigger*, *target places*, *vulnerable places* and an attribute. The attribute can be one of the simple primitives: computation, send, receive,

or guard. Therefore, the attribute is defined as an abstract class represented by *Attribute* and each primitive is defined as a sub-class, *Computation*, *Send*, *Receive*, or *Guard* respectively. A computation can be a normal computation (that is executed outside the ISR and updates the data state) represented by the sub-class *Compt*, an idle computation represented by the sub-class *Idle*, a delay (timer) represented by the sub-class *Delay*, or an exit computation (that raises an exception) represented by the sub-class *Exit*. We differentiate between these types of computation because each one has a different implementation. A guard can arise in three ways: as a result of a function call in the *case* statement, a value of a variable in the *case* statement, or an exception handled in the *trap-exit* statement. Each type of guard is defined by the sub-classes *GuardFun*, *GuardVar*, and *GuardExp* respectively because they are implemented in different ways.

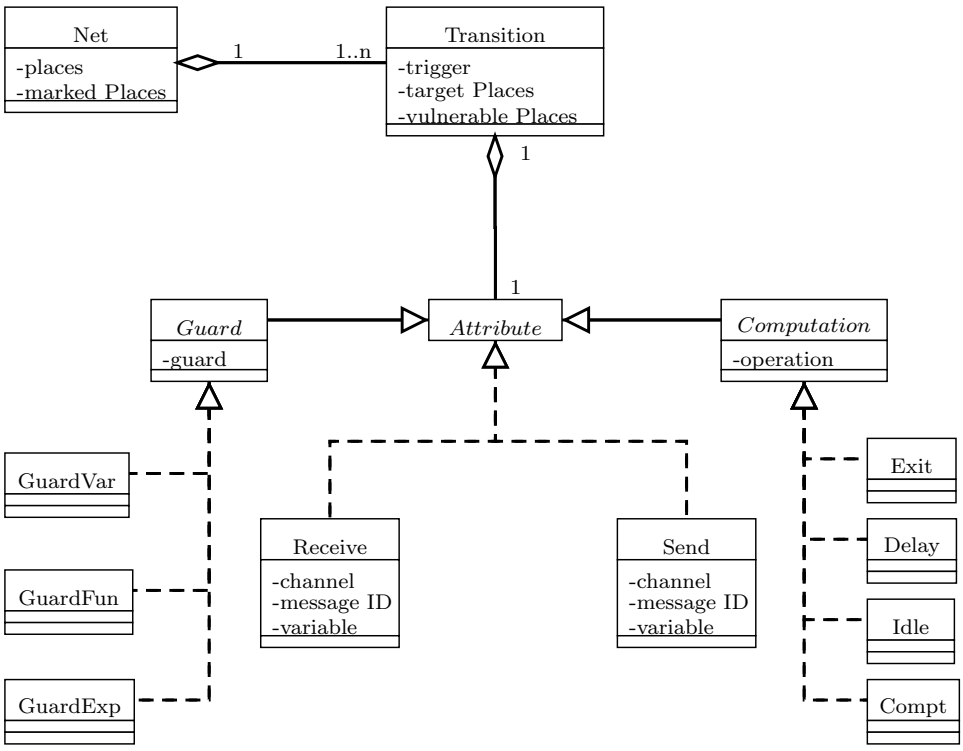


Fig. 3.7: Net architecture in UML.

In the following, we demonstrate how each element of the net is implemented.

First, we show the representation of the marked place, target places and vulnerable places. Next, the structure of the transition and the net is presented. After that, we illustrate the representation of the attribute. Then, we show how the data of the net is split between ROM and RAM. Finally, we give an example.

Places of the Net

The marked places, the target places and the vulnerable places of a net are represented by using *bit-set* data structures. A bit-set is defined as an array of words. A word is a platform dependent and could equal 8, 16, or 32 bits. There is a 1 or 0 corresponding to each element, specifying whether it is a member of the set or not. Bit-sets provide a more space efficient solution than normal arrays. Additionally, the C language features bit operations which can facilitate straightforward implementation of the set operations such as *intersection*, *union* and *complement*. These operations are needed to implement update marking of the net. The number of words needed to specify the size of the bit-set is expressed by `N_PLACE_WORDS`. The number of words of a 32-bit platform (for example) can be obtained from the total number of the net places `N_PLACES` using Equation 3.1.

$$\texttt{N\_PLACE\_WORDS} = \left\lceil \frac{\texttt{N\_PLACES}}{32} \right\rceil \tag{3.1}$$

Transition and Net Structure

A transition is represented as an ordinary C structure containing target places, vulnerable places and an attribute. The net is defined as an array of this structure. There is only one unique trigger (a number of a place) for every transition, therefore an index of a particular transition in the array expresses the trigger of the transition. Each transition must have a storage area for

an attribute. The Attribute is represented using three fields in the transition structure: `type`, `index`, and `attribute`. Fig. 3.8 shows the transition structure.

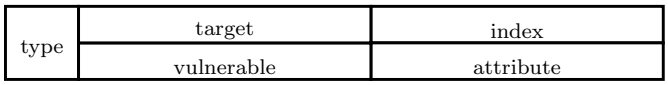


Fig. 3.8: Transition Structure.

Attribute Representation

The `type` $\in$ `AttributeType` indicates the type of an attribute. The set of attribute types is defined by:

$$\texttt{AttributeType} \mathrel{\widehat{=}} \{\texttt{IDLE, COMPT, DELAY, EXIT, GFUN, GVAR, GEXP, SEND, RECV}\},$$

where the elements of the set represent the sub-classes *Idle*, *Compt*, *Delay*, *Exit*, *GuardFun*, *GuardVar*, *GuardExp*, *Send*, and *Receive*, respectively. The `index` and `attribute` field are defined as integers and used in a variety of ways to store the details of different attributes. This is discussed in the following.

Computation When the attribute type is computation, there are three ways to deal with the attribute. First, if the computation is `IDLE` or `COMPT`, the `index` field of the transition refers to a particular block in a *net control block* (NCB) table. Each net has an NCB which represents the state of the net during the execution of the system. Primarily, the NCB deals with the case at which the net is currently executing a computation and so it indicates the status of the computation:

1. it is currently executing but not completed,
2. it is completed,
3. it is not currently executing.

The NCB also stores the result when the computation is a function. The `attribute` field of the transition contains the memory address of the C routine that implements the computation. Second, if the computation is `DELAY`, then the `index` field refers to a particular timer in a *timer table*. The timer table consists of a number of soft timers, and each timer is allocated to a delay transition. These timers are updated by the ISR in order to track the elapsed time for active (marked) delay transitions. The `attribute` field is used to store the number of ticks to delay. Third, if the computation is `EXIT`, then the `index` field refers to a particular block in an *exception control block* (ECB) table. Each net has an ECB used to indicate what exception is currently raised in the net. There is a flag associated with each exception in the net. The `attribute` field contains the exception mask which is used to access a particular flag in the ECB.

Guard When the attribute type is guard, there are three ways to deal with the attribute. First, if the guard is `GFUN`, then the `index` field refers to the NCB which stores the result of a function. The `attribute` field holds the intended value of the guard. Second, if the guard is `GVAR`, then the `index` field contains the memory address of the variable, and the `attribute` field also holds the intended value of the guard. Third, if the guard is `GEXP`, then the `index` field refers to an ECB in the ECB table. The `attribute` field contains the exception mask which is used to access to a particular flag in the ECB.

Send and Receive When the attribute type is `SEND` or `RECV`, the message identifier, message length, and an index to a particular block in a *port control block* (PCB) table, are packed in the `index` field. The PCB records the state of a particular port connected to a channel during execution. The PCB identifies whether the communication is *local* or *external*. It indicates if there is a new received message and provides a place to store

`type` field	`index` field	`attribute` field
`IDLE`	Index to NCB table.	Address of idle routine.
`COMPT`	Index to NCB table.	Address of computation routine.
`DELAY`	Index to timer table.	Number of ticks.
`EXIT`	Index to ECB table.	Exception mask.
`GFUN`	Index to NCB table.	Guard value.
`GVAR`	Address of variable.	Guard value.
`GEXP`	Index to ECB table.	Exception mask.
`SEND`	Message ID, message length, and index to PCB table.	Address of variable.
`RECV`	Message ID, message length, and index to PCB table.	Address of variable.

Tab. 3.1: Attribute representation summary.

the details of the received message including message identifier, message length, and data. Local and external communication is discussed later in section 3.4. The address of a variable that stores the message data, is stored in the `attribute` field.

The different representations of the attribute types in the transition structure are summarised in Table 3.1.

ROM or RAM

Constant data are not changed during run-time. This can be implemented in C by using the `const` qualifier. The `const` attribute allows a C compiler to place the constant data in ROM. The `transitions` (which is an array of transitions representing the net) is not changed during run-time, therefore it is more memory efficient to store it in ROM. Because `transitions` is constant variable, it must be initialised pre-run-time. The `marking` (which is a bit-set representing the current marking of the net) can be be changed during run-time, so it must be stored in RAM. The `marking` is defined as a usual variable in C.

Example

We use a modified version of the flow regulator example introduced in Chapter 2. We add a *trap-exit* and a *case* statement to the example in order to have all type of guards in the net representation. Additionally, the code from this example is generated for a single-node architecture, which means that both the `Flow` and `Valve` process run on the same node and communicate via a local channel. The aim of this example is not to describe a realistic system, but to generate a small-sized transition table that demonstrates the data structures of a net. Fig 3.9 shows the CANDLE program of the example. The process `Valve` is modified to raise the exception `ALARM` when the flow rate exceeds some range. Testing the flow rate is modelled using the `case testFlow()` statement. The function `testFlow()` can return three values: 0, 1, and 2 representing the state of the flow rate, *low*, *high*, or *out-of-range* respectively. The exception is handled by reporting a suitable warning and moving to an idle state thereafter.

```
Flow | Valve

where

Flow =
  every 10000 do
    readSensor();
    snd(k, FLOW, fFlow)
  end every
Valve =
  trap
     when ALARM => reportWarning(); idle
  in
    loop do
       rcv(k, FLOW, vFlow);
       case testFlow()
         when 0 => increaseFlow()
         when 1 => decreaseFlow()
         when 2 => exit ALARM
       end case
    end loop
  end trap
```

Fig. 3.9: Modified flow regulator in CANDLE.

The net representation of the example is shown in Fig. 3.10. In the figure, there are two nets, the top net represents the `Flow` process, and the bottom one represents the `Valve` process. The total number of transitions is 14, there is one trigger (place) for one transition. The idle transition is repeated to simplify the diagram. The places of the net are numbered to match their ordering in the generated transition table where transitions are sorted by their attribute type (in ascending order of transition number) as follows: send, receive, computation, guard, and delay. This order is adopted primarily for the correctness of the implementation (which is discussed in Chapter 4), and to improve the performance of operations (the ISR in particular) that work with the transition table.

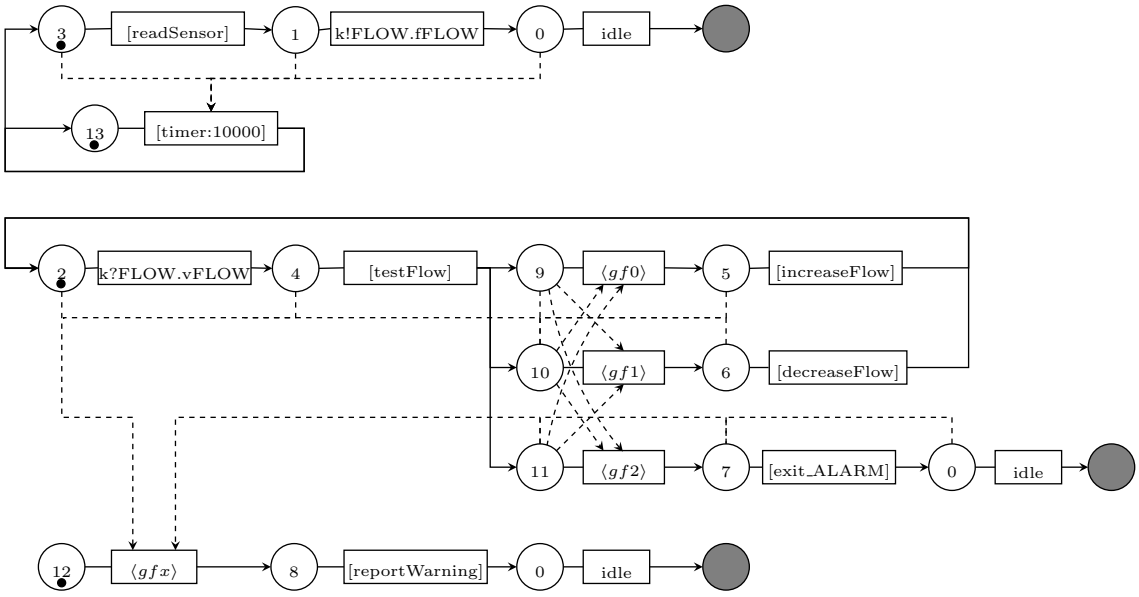


Fig. 3.10: Nets of the modified flow regulator example.

Fig. 3.11 shows the generated data structures of the example nets and the initial marking. There is a bit-set representing the current marking of the nets. The transition table comprises 14 transition structures representing the transitions of the nets. The NCB table comprises three blocks: one associated with each of the processes, `Flow` and `Valve`, and one associated with the idle process, which

executes when no other process has a computation ready to execute. The PCB table has one block because there is one port connected to a single local channel. The ECB table consists of three blocks similarly to the NCB table. There is only one delay transition in the nets so the timer table has one soft timer.

3.3.2 Overview of ISR Implementation

The state of a net is evolved (changed) according to two rules, the process transition rule $R.1$ and the network transition rule $R.2$ introduced in section 2.4.2. $R.1$ can modify both marking of the net and the data state, whereas $R.2$ can only modify the network state. We assume that the behaviour defined by $R.2$ is handled by the CAN controller. Therefore, only operations required by $R.1$ are considered for implementation in the ISR.

The ISR is configured to run periodically. The ISR updates the marking of a net according to the attribute type of a transition. The ISR implements the behaviour of the net as it is defined in the rule $R.1$. Algorithm 1 shows the pseudo code of the ISR. First, all active soft timers are updated in order to track the elapsed time. Next, the ISR polls all external receive buffers to record new arrived messages. If a message arrives after this step is completed, then it will only be considered in the next invocation of the ISR. After that, all local communication buffers are marked as stale (or reset). This ensures that any local message transmitted in the previous invocation of ISR will not be available again for reception in the current ISR. Then, the ISR reacts to all marked places in the net. In this step, a new marking of the net is obtained by firing all ready transitions. This step is repeated until the marking of the net becomes stable. This means that all ready instantaneous primitives (actions) are executed in the current ISR. However, in the first iteration of this loop, all external receive buffers are marked as stale. It means that any external message received at the beginning of the current ISR, will not be available again for reception in

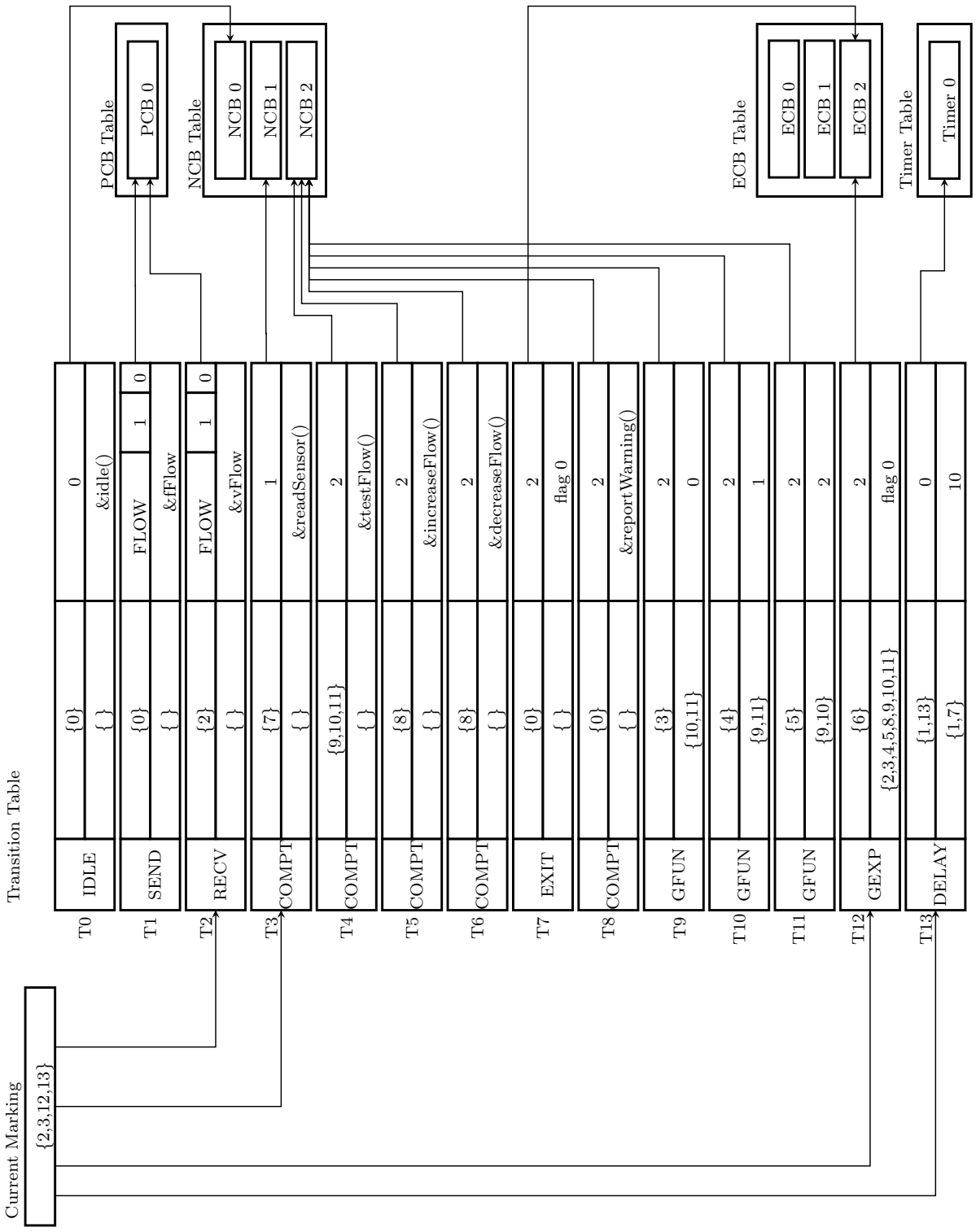


Fig. 3.11: Example of transition table.

next iteration, otherwise the ISR could enter an infinite loop. After that, one or more ready computations are scheduled to run during the next tick interval. Finally, the ISR makes all external messages ready for transmission just before it returns. In this step, the messages in the external transmit buffers become ready for transmission.

```
Update soft timers;
Update external port control blocks;
Mark all local port control blocks as stale;
repeat
    savedMarking ← marking;
    foreach marked place i do
        marking ← React(i);
    end
    if First iteration then
        Mark all external port control blocks as stale;
    end
until savedMarking = marking ;
Schedule next computation(s);
Flag external transmit buffers;
return
```

Algorithm 1: Pseudo code of the ISR.

The ISR reacts to a marked place according to the attribute type of the transition of which the place is the trigger. When the operation is completed, the current marking of the net is updated, and the system state (represented by the current values of NCBs, PCBs, ECBs) is modified. The pseudo code of the `React` operation is presented in Algorithm 2. When the condition of a particular attribute is satisfied, the transition is fired by executing the operation `Fire` and so the current marking of the net is updated. The pseudo code of `Fire` is shown in Algorithm 3. The operation consists of three steps: remove the source place p from the current marking, remove the vulnerable places $transitions[p].vulnerable$ from the current marking, and add the target places $transitions[p].target$ to the current marking.

```
Input: A transition trigger, p.
Output: Updated marking and system state.
switch transitions[p].type do
    case IDLE
        Skip;
    end
    case COMPT
        if computation is ready and not completed then Skip;
        else if computation is completed then Fire(p);
        else if computation is not ready then Mark as ready;
    end
    case DELAY
        if timer is active then
            if timer is expired then
                Fire(p);
                Make timer inactive;
            end
        end
        else if timer is inactive then
            Load a delay value to timer;
            Make timer active;
        end
    end
    case EXIT
        Raise the corresponding exception in ECB;
        Fire(p);
    end
    case GFUN
        if Function result in corresponding NCB = guard value then
            Fire(p);
        end
    end
    case GVAR
        if The variable value = guard value then
            Fire(p);
        end
    end
    case GEXP
        if The exception is raised in corresponding ECB then
            Clear the exception flag in corresponding ECB;
            Fire(p);
        end
    end
    case SEND
        if External channel then
            Write message details to an external transmit buffer;
            Fire(p);
        end
        else if Local channel then
            Write message details to corresponding PCB;
            Fire(p);
        end
    end
    case RECV
        if Fresh message available in corresponding PCB then
            if message id is matched then
                copy the message content to a user data variable;
                Fire(p);
            end
        end
    end
end
return
```

Algorithm 2: Pseudo code of `React`.

Input: A transition trigger, p and the current marking, $marking$ of the net.
Output: New marking, $marking'$ of the net.
$marking' = marking \setminus (\{p\} \cup transitions[p].vulnerable) \cup transitions[p].target$
return

Algorithm 3: Pseudo code of `Fire`.

3.4 Implementation of Channel

A channel is an abstraction of the CAN protocol through which processes communicate via asynchronously broadcast messages. All communications between system components occur through this mechanism and never through the use of shared variables. A message is identified with the pair (id, val). The id is the message identifier which corresponds to the message priority in CAN. The val is the value of the message which represents the data field of the CAN frame.

bCANDLE processes can be allocated to a number of nodes which are connected by one or more CAN buses. The allocation of processes determines the implementation of channels. When the process A communicates with the process B by the channel k and both A and B are allocated to the same node, then k is usually mapped to a *local channel*. In contrast, when A and B are allocated to two separate nodes, then k is mapped to an *external channel*. Fig. 3.12 shows four examples of the possible process-to-node mapping. In the case a, the processes $P1$, $P2$ and $P3$ are completely distributed, so they must communicate using an external channel. In the case b, all processes share the same node, therefore a local channel must be used for communication. In the case c, the process $P1$ and $P2$ are allocated to the same node, whereas the process $P3$ is allocated to another node. The processes, in this scenario, must communicate by using an external channel since there is at least one process that is allocated to a separate node. In this particular case, although $P1$ and $P2$ are allocated to the same node, the communication between the two processes occurs through an external channel. This requires that the communication controller has the ability to be configured for a self-reception in which a transmitted message can

be received by the same node. For example, the CAN controller of Philips LPC2294 supports this feature. Finally, the case d shows that the processes $P1$, $P2$, $P3$, and $P4$ are allocated to the same node, then they must use a local channel for communication. However, the process $P5$ is allocated to a separate node, then $P5$ and $P4$ must communicate using an external channel.

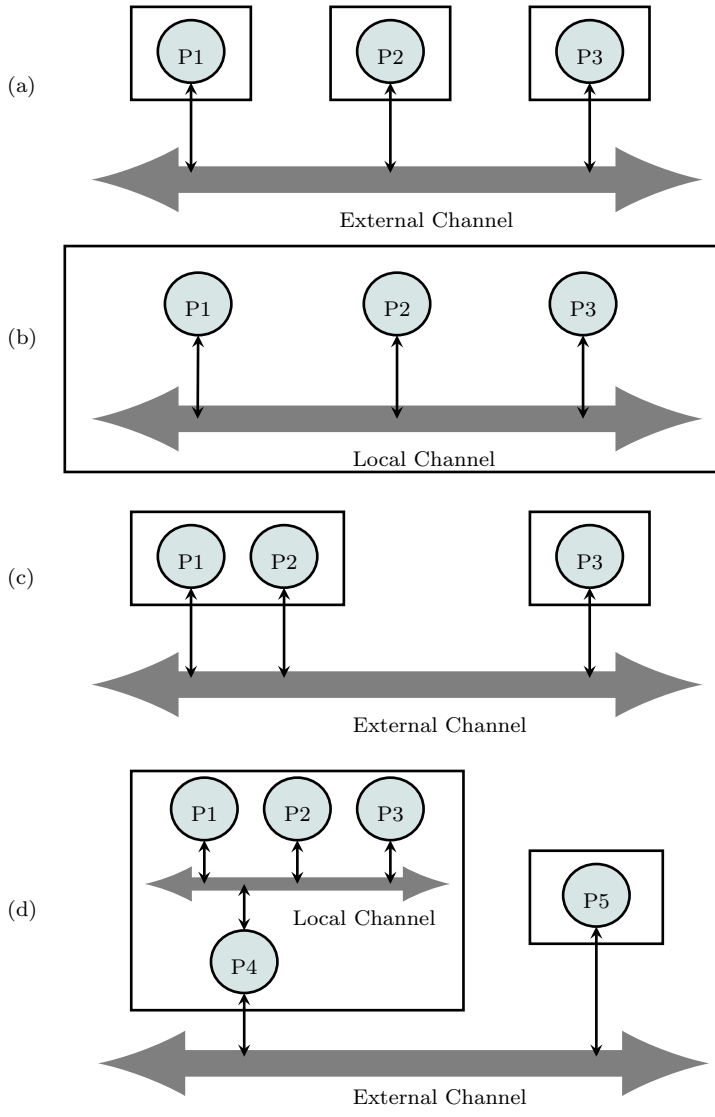


Fig. 3.12: Processes-to-nodes mapping.

Two ways of implementing a channel are introduced: external channel and local channel. We use a simple send/receive example to demonstrate the two different

implementations of channels. In the example, process $P1$ broadcasts a message $(i, var1)$ through the channel k and idles thereafter. The process $P2$ is pending on channel k until it receives a message with the identifier i. The content of the message is held in the variable $var2$. Fig. 3.13 shows bCANDLE model of the example, and Fig. 3.14 shows the net representation of the model.

```
P1 | P2
where
P1 = k!i.var1 ; idle
P2 = k?i.var2 ; idle

network
k = (i: pri, lb, ub, lB, uB)

data var1, var2
```

Fig. 3.13: The bCANDLE model of simple send/receive example.

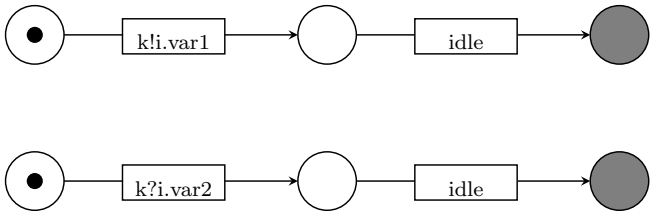


Fig. 3.14: The Net of simple send/receive example.

3.4.1 External Channel

When the communicating process $P1$ and $P2$ are allocated to two separate nodes, then the channel k must be mapped onto a physical communication link. The processor delegates responsibility for the communication to an external CAN controller. Fig. 3.15 illustrates the structure of a communication between processes $P1$ and $P2$. When $k!i.var1$ of the process $P1$ is marked and ready to run, the ISR writes the message content held in the variable $var1$ to a CAN transmit buffer. The ISR then sets a flag $transmit\_flag$ to indicate that the message is available for transmission. On the receiving node, When $k?i.var2$ of $P2$ is marked and ready to run, the ISR polls a flag $receive\_flag$

to check the arrival of the message. This occurs periodically until the intended message arrives. When a message is received with the identifier i, then the ISR reads a CAN receive buffer and assigns the message content to the variable $var2$ of the process $P2$.

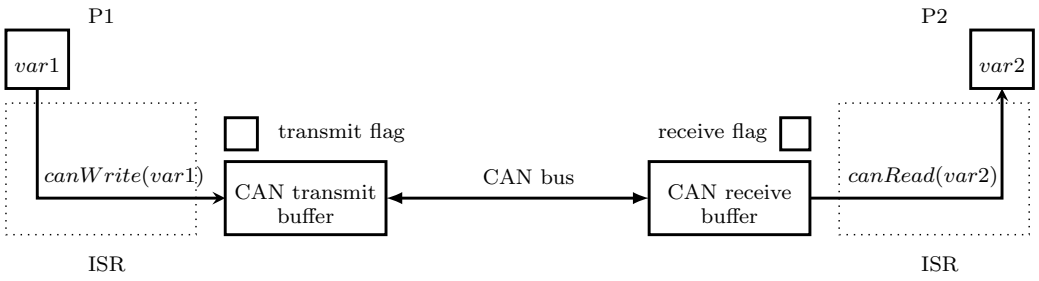


Fig. 3.15: P1 communicates with P2 by external channel.

3.4.2 Local Channel

If the process $P1$ and $P2$ are allocated to the same computing node, then the channel k is mapped onto a local channel. When $k!i.var1$ of the process $P1$ and $k?i.var2$ of $P2$ are both marked and ready to run, the memory address of $var1$ and $var2$ are passed to the ISR. The ISR then copies the value of $var1$ to $var2$ provided both the source and destination message identifiers are matched, see Fig. 3.16. The current example has only one receiver process. However, when there are a number of receiver processes, the ISR will provide each receiver a copy of the value of the sent message at the same time.

3.5 Representation of the Architecture

The code generator requires information about the architecture of the execution platform in order to produce the code correctly. This includes information about processes, nodes, process-to-node allocation, scheduling algorithm, tick

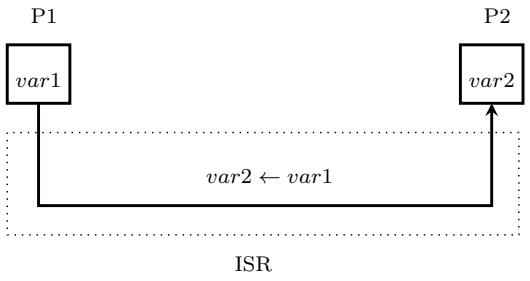


Fig. 3.16: P1 communicates with P2 by local channel.

rate, and communication details (e.g, network transmission rate and message identifiers). In the current work, a special-adopted file format is used to describe the architecture of a system. The adopted language has a simple textual syntax but allows to specify all details which are necessary for the code generation.

Fig.3.17 shows an example of a configuration file created for the flow regulator example. The system consists of two computing nodes: `flow` and `valve`. The process `Flow` is allocated to the node `flow` and `Valve` is allocated to the node `valve`. The first node uses 32 bits word size, has 1000μsec tick rate, employs a simple round-robin scheduling algorithm, and defines a single external communication port. There is a separate section to define processes. For instance, the process `Flow` allows 20 words at maximum for the stack size and defines one channel `k` mapped to the communication port `CAN_0`. Additionally, there is a section to describe the communication channels. The section defines the transmission rate and messages exchanged between the system components. For instance, in the example shown in Fig 3.17, the transmission rate is set at 100 $kbit/s$. Messages with the identifier `FLOW` are used for communications, and the size of the data field of a message is only 1 byte.

There is however an industry standard language to specify a system architecture. The Architecture Analysis and Design Language (AADL) (Feiler et al., 2006) can be used to describe the execution platform of the generated code. The language has both a textual and a graphical representation. The AADL

```
node flow
  wordsize   : 32
  tickHz     : 1000
  processes  : Flow
  scheduler  : ROUND_ROBIN
  ports      : CAN_0

process Flow
  stacksize : 20
  channels  : k -> CAN_0

node valve
  wordsize   : 32
  tickHz     : 1000
  processes  : Valve
  scheduler  : ROUND_ROBIN
  ports      : CAN_0

process Valve
  stacksize : 20
  channels  : k -> CAN_0

channel k
  bps : 100000
  messages : <FLOW:1>
```

Fig. 3.17: The architecture description of the flow regulator example.

comprises elements to specify software components (*data*, *thread*, *process* and *subprogram*), hardware components (*device*, *memory*, *bus* and *processor*) and composition (*system*).

Fig. 3.18 shows the AADL representation of the flow regulator example. The processes `Flow` and `Valve` are bound to the nodes `flow` and `valve` respectively. The channel `k` is an external channel and therefore it is bound to the physical CAN bus `CAN_0`. We use the `system` element to represent a channel. This element is abstract and can be mapped to a software or to a hardware component. Therefore, it is suitable for specifying both external and local channels.

The second example shown in Fig 3.19 presents another configuration for the flow regulator example. The processes `Flow` and `Valve` are bound to the same node `flow−valve`. The channel `k` is local channel and so it is bound to a `memory` element which represents the local communication.

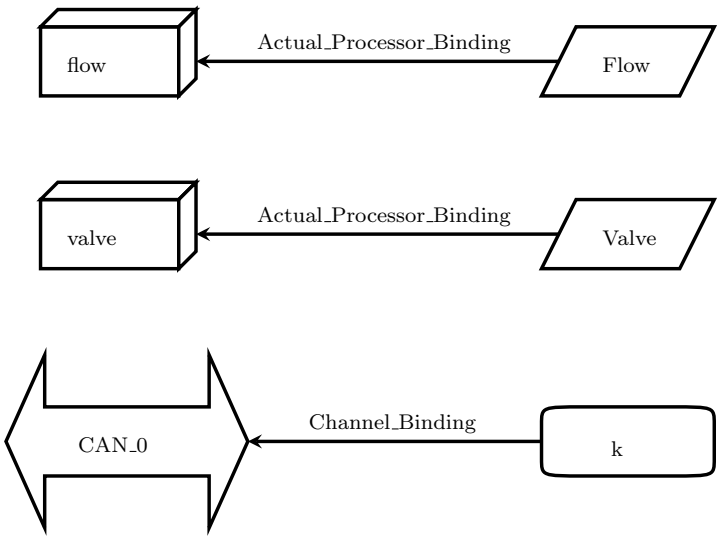


Fig. 3.18: The AADL of the flow regulator example (distributed architecture).

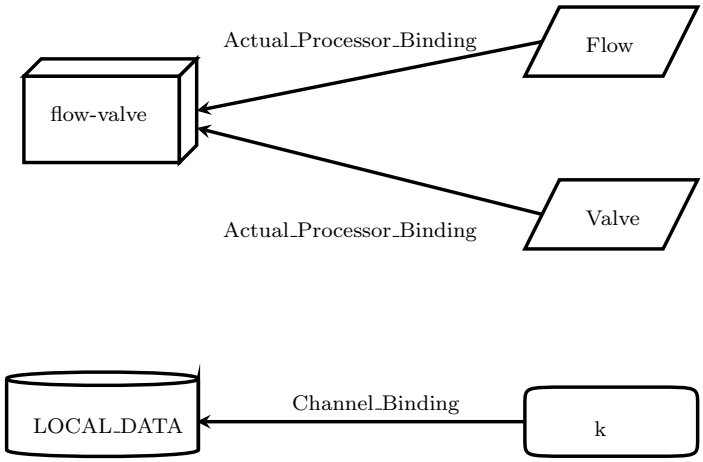


Fig. 3.19: The AADL of the flow regulator example (single-node architecture).

AADL has built-in properties to provide information about system components. A property has a name, type, and an associated value. For instance, the property *Actual_Processor_Binding* allows to bind a process to a processor component. Interestingly, the language permits defining new properties. This feature is very useful in specifying other system information of node components (e.g, word size, scheduling algorithm, tick rate), process components (stack size and channel binding), and channel components (e.g, transmission rate and message

identifiers). This can be a subject of future work.

3.6 Summary

The CANDLE code generator has been introduced in this chapter. Executable code is derived from a net which is the intermediate model of a CANDLE system. Our implementation model to execute the net has been presented. An efficient C representation of the net has been designed. The implementation of broadcast channels has been described. Finally, an AADL-like language has been adopted to describe the system architecture including nodes, processes and channels. The approach taken simplifies the process of reasoning about the relationship between generated formal model and its implementation, because both the model and code are derived from the same net. The main point of this chapter is to generate executable code for CAN-based distributed embedded systems in a way that guarantees that both functional and timing properties expressed in a high-level formal language are satisfied. Chapter 4 presents a rigorous argument demonstrating that a generated formal model is a conservative approximation of its implementation. Chapter 6 demonstrates that the approach adopted provides an efficient implementation. To the best of our knowledge, this is the first work to apply such a method to the implementation of distributed embedded systems.

4. CORRECTNESS OF SYSTEM IMPLEMENTATION

4.1 Introduction

The main goal of this work is to produce a method for the development of CAN-based embedded systems that provides both for the generation of reasonably efficient system implementations and also formal models that conservatively approximate their implementations; both components are to be generated automatically from the same system description. This chapter addresses the implementation and modelling decisions that have been made to ensure that models conservatively approximate their implementations.

We have adopted the time-triggered implementation model which has been described in Chapter 3. According to the implementation model, all instantaneous primitives are assumed to run inside the ISR. Computation primitives however are assumed to run outside the ISR. In this chapter, we discuss the correctness of the implementation of each primitive. An informal argument has been used to demonstrate that the implementation satisfies the semantics of its model. The reason for this approach is that the target implementation language (which is C) lacks a formal semantics. Although there have been many attempts to provide a formal semantics to C, for example recently in (Ellison and Rosu, 2012), the validation of the translation from C to a low-level machine language is very important to ensure that the C compiler generates an executable code

that behaves exactly as specified by the semantics of the source program. A rigorous solution to this problem is not trivial. The problem has been addressed in the *CompCert* project (CompCert, 2012) and the results have been extensively published, for instance (Blazy, 2008; Dargaye, 2009; Leroy, 2009; Blazy and Leroy, 2009; Bedin et al., 2012).

The chapter is organised as follows. Sections 4.2, 4.3, 4.4 and 4.5 discuss the modelling and implementation decisions made to ensure the correctness of the formal language primitives, including computation, guard evaluation, message reception, and message transmission respectively. Section 4.6 discusses the correctness of the process combinators (sequential composition, choice, interrupt, and parallel composition). Finally, section 4.7 concludes the chapter.

4.2 Computation Release and Termination

In this section we consider how to calculate lower and upper execution time bounds for the models of computations released by a process P that is assumed to be the only user process allocated to its node. The bounds are calculated to ensure that the model is a conservative approximation of the implementation. We begin by distinguishing between the sets of *observable* computations, $\mathcal{O}$, e.g. computations that interact with the environment, perhaps by reading from a sensor or writing to an actuator, and *internal* (unobservable) computations, $\mathcal{I}$, that merely update the local data state without interacting with the environment. We assume the set of all computations, $\mathcal{C} = \mathcal{O} \cup \mathcal{I}$ and $\mathcal{O} \cap \mathcal{I} = \emptyset$. The construction of models for internal computations is considered first.

4.2.1 Internal Computations

Fig. 4.1 shows the implementation structure and its related model for a process fragment given by $P = \ldots C_1(); C_2(); \ldots$, where $C_1, C_2 \in \mathcal{I}$.

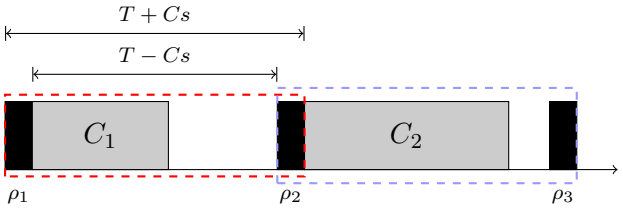


Fig. 4.1: Internal computation bounds $(we(C_1) <= \chi)$ $[C_1 : T - Cs, T + Cs]; [C_2 : T - Cs, T + Cs]$.

The worst-case (resp. best-case) execution time of a computation C is denoted $we(C)$ (resp. $be(C)$). We assume in the case of both computations C_1 and C_2 that their worst-case execution times are no greater than the length of a computation interval. Hence we can be sure that their actual execution times, e_1 and e_2 respectively, are within the computation time available in one tick, i.e. $e_1 <= \chi$ and $e_2 <= \chi$. Notice that the model $[C_1 : T - Cs, T + Cs]; [C_2 : T - Cs, T + Cs]$ allocates bounds enclosing a full tick interval to each computation in order to account both for any idle time until the next tick and also the time taken for execution of the ISR. This is similar to the standard approach taken to account for the time used by a tick scheduler in the response time analysis of fixed priority systems, e.g. as described by Liu (Liu, 2000). Fig. 4.1 shows the actual execution time as a shaded grey box and the upper bound on the extended time as a dashed box. The lower and upper bounds are chosen so that the model contains behaviours that include successive reaction instants ρ_1 and ρ_2 in which ρ_1 occurs at the end of the first ISR and ρ_2 occurs at the beginning of the second ISR (lower bound: $T - Cs$), and, conversely, ρ_1 occurs at the beginning of the first ISR and ρ_2 occurs at the end of the second ISR (upper bound: $T + Cs$). Therefore, we can be sure that the model contains all possible behaviours of the implementation. In the case that a computation requires more than one tick to complete, the bounds on its completion time are calculated accordingly. Fig. 4.2 illustrates a computation requiring more than one tick to complete and shows its associated model. Once again, the bounds

calculated give rise to a non-deterministic model that allows the notional time of a reaction instant to occur at any point within an ISR.

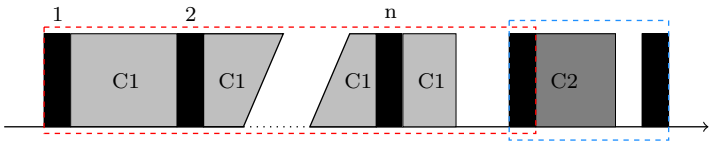


Fig. 4.2: Internal computation bounds $(we(C_1) > \chi)$
$[C_1 : n.T - Cs, n.T + Cs]; [C_2 : T - Cs, T + Cs]$.

4.2.2 Observable Computations

We assume that all observable computations complete their execution within a single computation interval. Even so, the model of an observable computation is a little trickier to construct than that of an internal computation, since we need to consider its termination not only with regard to the time of the next reaction instant but also with regard to the time of its interaction with the environment. Fig. 4.3 shows the implementation structure and its related model for a process fragment given by $P = \ldots C_1(); C_2(); \ldots$, where $C_1 \in \mathcal{O}$ and $C_2 \in \mathcal{I}$.

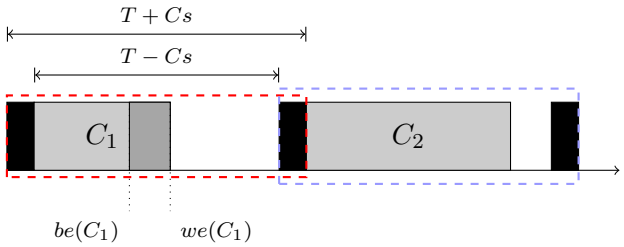


Fig. 4.3: Observable computation bounds $(C_1 \in \mathcal{O}, C_2 \in \mathcal{I})$
$([C_1 : be(C_1), we(C_1)]; idle\,[> [T - Cs, T + Cs]); [C_2 : T - Cs, T + Cs]$.

The associated model is constructed to capture both the actual termination time of C_1, since it may be observed by the environment, and also the effective termination time of C_1 in so far as it causes the release of C_2 at the next reaction instant. The first part is modelled by $[C_1 : be(C_1), we(C_1)]; idle$, i.e. C_1 actually terminates, and may be observed, at some time between its best-

case and worst-case execution times; then the idle computation executes until the next reaction instant. The timing of the next reaction instant is within the usual bounds for a computation executed within one tick period and is modelled by the interruption of the idle computation by a non-deterministic timeout, $[T - Cs, T + Cs]$, whose termination allows the release of C_2. So the complete model in this case is

$$([C_1 : be(C_1), we(C_1)]; idle\ [> [T - Cs, T + Cs]); [C_2 : T - Cs, T + Cs]$$

and, in general, the model of any observable computation C is just

$$([C : be(C), we(C)]; idle\ [> [T - Cs, T + Cs])$$

4.2.3 Delay

Explicit delays are introduced into system descriptions using the *elapse*(<*timeSpec*>) statement, where *timeSpec* is a time specification, e.g. *seconds(5)*, *milliseconds(10)* etc. The code-generator converts *timeSpec* into a number of ticks and the model-generator generates a model in which the *elapse* statement is represented as an internal computation that implements the identity operation in a time whose bounds are calculated as in section 4.2.1.

For example, consider the implementation of the statement *elapse(microseconds(5500))* on a node with a tick period $T = 2\ ms$. The time specification is converted into a number of ticks as follows:

$$microseconds(5500) = \left\lceil \frac{5500 \cdot 10^{-6}}{2 \cdot 10^{-3}} \right\rceil = 3$$

and the delay is modelled as $[3T - Cs, 3T + Cs]$, as shown in Fig. 4.4.

Notice that if the requested delay is not a multiple of the tick period, the implementation provides a delay that is close to the least multiple of the tick

Fig. 4.4: Delay bounds ($elapse(microseconds(5500))$, $T = 2\ ms$, $[3T - Cs, 3T + Cs]$.

period greater than the requested delay. The generated model represents this.

4.2.4 Response-Time Analysis

We must be able to perform an offline calculation of the bounds on the response time of all computations for any chosen scheduling algorithm in order to generate a model. In the case of a computation, C, released by a cooperatively scheduled process, we know that it will complete within one tick and so its model is given simply as $[C : T - Cs, T + Cs]$, as illustrated in Fig. 4.1. Computations that cannot be scheduled in a single tick are called *long-running computations* and are scheduled using a weighted round-robin algorithm. The remainder of this section considers how to calculate the bounds on such computations.

The number of ticks required to provide e time units of execution for a long-running computation is given by

$$nt(e) = \left\lceil \frac{e}{\chi_{rr}} \right\rceil \tag{4.1}$$

where χ_{rr} is the length of time allocated for the execution of round-robin processes in each tick period, i.e. in the hybrid scheduling algorithm the length of a round-robin slot is given by χ_{rr}

If a set of processes $\{P_1, \ldots, P_n\}$ is sharing the CPU using a weighted round-robin scheduler, a computation of length e_i released by process P_i will require a number of rounds given by $\lceil \frac{nt(e_i)}{s_i} \rceil$, where s_i gives the weight of P_i equal to the number of round-robin slots allocated to it in a complete round of the schedule.

For example, let the set of processes and their weights be as follows:

$$\{(P_1, 1), (P_2, 2), (P_3, 3), (P_4, 4)\}$$

Then every round is of length 10 and in each round a process is allocated a number of slots according to its weight. A round in this case would be

$$P_1, P_2, P_2, P_3, P_3, P_3, P_4, P_4, P_4, P_4,$$

and process P_3, for example, would have $3\chi_{rr}$ time units allocated to it in every round for the execution of its long-running computations. □

Worst-case response time

The worst case response time for a long-running computation C_i, released by process P_i, occurs when C_i is released just as the scheduler is about to schedule P_{i+1} for the first time in this round, i.e. when P_i must wait for all other processes to use their full allocation and for its slot in the next round to arrive before it can begin execution.

From this we conclude that the number of ticks required in the worst case to complete a long-running computation of length e_i, released by P_i, in a round of length R is given by

$$wt(e_i) = \left(R \cdot \left\lceil \frac{nt(e_i)}{s_i} \right\rceil\right) - \left(\left\lceil \frac{nt(e_i)}{s_i} \right\rceil \cdot s_i - nt(e_i)\right) \tag{4.2}$$

The term on the left of the minus sign gives the number of rounds required for the computation multiplied by the length of a round. The term on the right compensates for any over-allocation of slots computed by the first term.

For example, assume the same set of processes and weights as in the example above and a round-robin allocation $\chi_{rr} = 500$ time units. Now, assume that

process P_3 releases a computation with a worst-case execution time of 1800 time units. This will require $\lceil \frac{1800}{500} \rceil = 4$ ticks which it will receive in $\lceil \frac{4}{3} \rceil = 2$ rounds. Each round is of length 10, so we can say that the computation will complete in no more than 20 ticks. Actually, in 2 rounds P_3 will receive 6 ticks, i.e. 2 ticks more than needed for this computation. So, in fact, the computation will complete in 18 ticks, as given by equation 4.2. □

Having calculated the number of ticks required in the worst case for the completion of a computation, C_i, the worst-case response time of C_i is given simply by the product of this value and the length of the tick period.

$$W(C_i) = T \cdot wt(we(C_i)) \tag{4.3}$$

Best-case response time

In the best case the scheduler will be just about to schedule P_i for the first time in this round, so it will not be necessary for P_i to wait for the rest of the round to begin its slot. This gives us a formula for the number of ticks required in the best case to complete a long-running computation of length e_i, released by P_i, in a round of length R:

$$bt(e_i) = wt(e_i) - \sum_{j \in \{1,\ldots,n\} \setminus \{i\}} s_j \tag{4.4}$$

So the number of ticks required in the best case is the just the number of ticks required in the worst case minus the number of ticks required for the rest of the round, i.e. the number of ticks allocated to all processes except P_i. The best-case response time of a computation C_i is then given by

$$B(C_i) = T \cdot bt(be(C_i)) \tag{4.5}$$

4.3 Guard Evaluation

A guard is the name of a predicate evaluated in the current data environment which gives a value either *true* or *false*. There are two ways in which a guard can arise. First, in the CANDLE statement `case`, the result of a function call (or the value of a variable) is compared with a number of constant values. Second, an exception is raised when the exit statement is executed in the `trap − exit` statement. In each case the guard is handled (evaluated) inside the ISR which is performed as a simple test of one value (the result of the evaluation) with another (e.g, a particular case expression value). The evaluation of a predicate for a function call occurs as a computation executed outside the ISR. Then the ISR compares the result of the computation with a simple value. In the case of exception, a flag is set inside the ISR to raise an exception. Then the ISR performs a simple test to see whether an exception flag is set or not.

4.4 Message Reception

Particular care needs to be taken to ensure that the generated model is a conservative approximation of its implementation in the case of message reception. There are three main requirements:

1. to ensure that there is an adequate number of receive buffers to store messages transmitted between polling operations so that messages are not lost;

2. to identify precisely when a node is marked as ready for message reception and to account for the time between polling operations; and finally

3. to identify the order of message receptions at different CAN controllers in gateway nodes.

4.4.1 Receive Buffers

If there are n receive buffers in a CAN transceiver then

1. there should be no more than n possible acceptances in any interval that is no longer than $T + Cs$, and
2. the receive buffers should be flushed at each reaction in the ISR.

These requirements ensure that

1. messages are not lost, and
2. the time to react to a message is properly accounted for.

4.4.2 Reception Readiness

Fig. 4.5 illustrates the difficulty in ensuring that messages are received only when a node is clearly ready for their reception, according to the semantics of the model. The figure shows the behaviour of a node with a single process executing the process fragment $\ldots C_1(); rcv(k, i, x); C_2() \ldots$, in three different scenarios for the transmission of a message $i.v$ on the CAN bus. The questions are in which of the scenarios should the message be received and how is the timing of the reception modelled.

First, remember in our time-triggered implementation model, CAN messages are polled in each ISR. This is the only opportunity for message reception in an implementation. At ISR 2 in Fig. 4.5, a message will be available for reception in scenarios Bus (1) and Bus (2). However, in the former case, the actual execution of computation C_1 will not have completed before the message acceptance point occurs, and so clearly message reception $k?i.x$ cannot be enabled in time; and in the latter case, although the actual execution of C_1 will have completed before the message acceptance point, its model represents its termination as

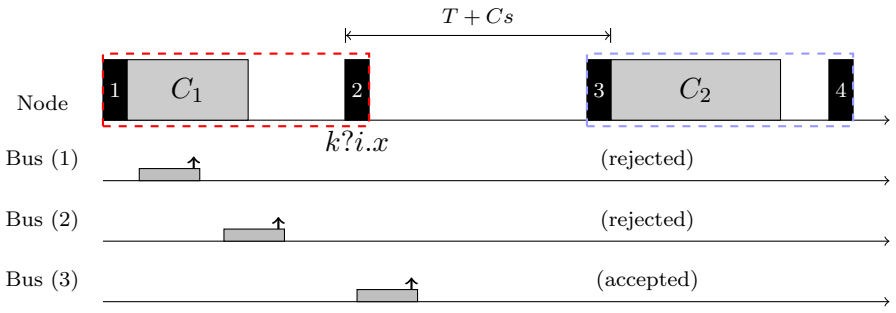


Fig. 4.5: Message reception (↑ denotes the message acceptance point) $[C_1 : T - Cs, T + Cs]; k?i.x; [0, T + Cs]; [C_2 : T - Cs, T + Cs]$.

occurring at the reaction instant in ISR 2, and again $k?i.x$ is enabled after the message acceptance point. So, in both of these scenarios, the semantics of the model requires that the message should not be received. In scenario Bus (3), the reaction instant at ISR 2, when $k?i.x$ is enabled, is clearly not later than the message acceptance point. In this case, the message should be received at ISR 3 and computation C_2 will be released in the next computation interval.

It is clear that, at ISR 3, the model needs to allow for the message acceptance to have occurred at any time between the reaction instants at ISR 2 and ISR 3. It is not possible to tell, a priori, precisely when the message acceptance point will occur. Hence, the inclusion in the model of the non-deterministic delay $[0, T + Cs]$.

The approach taken to ensure that the implementation corresponds with the semantics of the model is to order the transition table so that receive transitions are considered first (before computation terminations) and, after the first pass through the reaction loop, external messages are marked as 'stale', i.e. no longer available for reception in the current reaction. Following this procedure, it can be seen that a message will not be received in scenarios Bus (1) and Bus (2) but that a message will be received in scenario Bus (3).

4.4.3 Reception Order

A node may be configured with a number of different CAN controllers. This could happen in gateway nodes. Fig. 4.6 shows the behaviour of a gateway node with a single process executing the process fragment

$$k_1?i_1.x; [0, T + Cs]; \;\; [C_1 : T - Cs, T + Cs]$$
$$+$$
$$k_2?i_2.y; [0, T + Cs]; \;\; [C_2 : T - Cs, T + Cs]$$

and two messages from two different CAN buses arrive in the same interval. According to the model both $k_1?i_1.x$ and $k_2?i_2.y$ are enabled. So if the message i_1 is received first then the computation C_1 is released and $k_2?i_2.y$ is disabled. However, if the message i_2 is received first then the computation C_2 is released and $k_1?i_1.x$ is disabled. When the acceptance point of the messages occurs in the same interval, then both message i_1 and i_2 become available for reception. The difficulty in the implementation is to determine which message should be polled first inside the ISR. There are three possible cases to consider:

1. both channels are external,
2. one channel is external and one is local,
3. and, both channels are local.

A possible solution to the first case can be achieved by using CAN controllers that provide a time stamp on received and transmitted messages (e.g, (Texas-Instruments, 2005)). When the received messages are time stamped, then the ISR can identify the order of message receptions and update the transitions table accordingly.

For the second case, the order of message receptions is ensured by the ISR implementation. The ISR polls the external receive buffer(s) and updates any

ready receive transition before checking the local communications. Therefore, if a message acceptance occurs during the previous interval, the ISR considers that the external message arrives first. However, if the message acceptance occurs after the beginning of the ISR, then the local message will be considered first and the external message will not be processed until the next ISR runs.

In the third case, the order of message receptions is resolved non-deterministically. Local communication occurs inside the ISR where the transmission and reception of a message are assumed to happen instantaneously. Therefore, any order is chosen in the implementation would be acceptable since the model can express both behaviours.

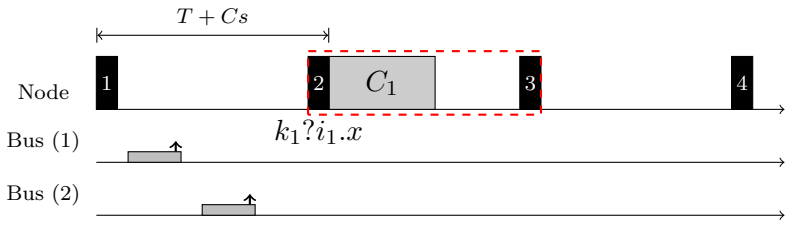


Fig. 4.6: Example of message reception in a gateway node.

Receive with Interrupt Operator

Consider the following process fragment:

$$k?i.x; [0, T + Cs]; [C_1 : t] \ [> \ [Timer : t]; [C_2 : t]$$

The new data value of the variable x only becomes available after $[0, T + Cs]$ is completed. According to the semantics of the interrupt operator, the first argument may be interrupted at any instant once it is enabled. If it is interrupted just before $[0, T + Cs]$ begins, then the new state of x must be visible based on the semantics of $k?i.x$. However, the $[Timer : t]$ is a time delay and is only evaluated inside the ISR. The message is received and the variable is updated inside the ISR as well. If we ensure this happen before evaluating the

timer, then any other primitive will see the new state of the variable after it is updated. So the term $k?i.x;[0,T+Cs]$ behaves atomically to the second argument of the interrupt operator.

Similarly if a guard $<\gamma>$ appears in the second argument. In this case, we have to ensure also the guard evaluation is performed after the message is completely received inside the ISR.

Moreover, a message reception may appear in the second argument. There are two possibilities: the two receive statements read from the same channel, or the two receive statements read from two different channels.

For the first case, one can consider the following example:

$$k?i1.x;[0,T+Cs];[C_1:t] \;\; [> \;\; k?i2.y;[0,T+Cs];[C_2:t]$$

Now, if the channel k is local, then the evaluation of the two ready receive statements can be performed non-deterministically because the local communication occurs instantaneously inside the ISR.

When the channel k is external, the evaluation is performed as follows. If $i1$-message is received before $i2$-message, then the ISR polls $i1$ and updates x before processing $i2$. This can be guaranteed by the employed CAN controller. For example, Motorola msCAN08 utilises a double receive buffers. The incoming messages are stored in a two stage input FIFO (MC68HC08, 2012). The first received message is stored in a *foreground* buffer and so it becomes available for a polling software component. The second received message is stored in a *background* buffer and it only becomes available once the foreground buffer is read. Therefore, once $k?i1.x$ is started, it can only be interrupted after $[0,T+Cs]$ is completed. On the other hand, when $i2$-message is received first, then the ISR polls $i2$ and updates y before processing $i1$. So $k?i1.x$ is interrupted before it begins. Consequently $k?i1.x;[0,T+Cs]$ is either executed at once or

interrupted before it is started, i.e. the term behaves atomically to the second argument of the interrupt operator.

In the second case, one can consider the following example:

$$k1?i1.x;[0,T+Cs];[C_1:t] \;\; [> \;\; k2?i2.y;[0,T+Cs];[C_2:t]$$

The evaluation of the message reception is resolved in a way that is similar to the order of message reception discussed in section 4.4.3.

4.5 Message Transmission

According to our implementation model, transmission of a message is initiated inside the ISR which is assumed to happen instantaneously. In this section we discuss two issues that could affect the implementation correctness of the message transmission.

4.5.1 Transmission Readiness

The implementation requires a non-zero time before a message becomes ready for transmission. If the channel is external, the ISR accommodates the message contents for any active $k!i.x$ transition in an available transmitter buffer. Then it clears a transmitter flag to indicate that the message is ready for transmission and fires the transitions. When the channel is local, the ISR transfers the message contents of any active $k!i.x$ transition into an intermediate buffer and then fires the transitions. The operation of message transmission may be completed at any instant in between the beginning and the end of the ISR.

If we assume that the message is enqueued for external transmission at the beginning of the ISR, then the model may exhibit a behaviour where a message is enqueued and the bus is free. In this case the message can be transmitted

without contention for the bus. In practice, the message may be enqueued at any instant up to the end of the ISR. The problem is that the bus could be occupied by another message and so the message has to wait until the bus becomes free again. Therefore, this possible behaviour of the implementation is not expressed in the model. Similarly if we assume that the message is enqueued at the end the ISR, then it is possible to find a case where the implementation exhibits a behaviour that is not expressed in the model.

The problem is that the message is allowed to be enqueued at any moment during the ISR time. If this is prevented, then the implementation would only have the same behaviour. To resolve this problem, the ISR should accommodate the message contents in the transmitter buffer and clear the transmitter flag just before the ISR terminates. Then the message can only contend for the bus at the end the ISR.

When the message is enqueued for local transmission, the two assumptions of the ISR can be considered. Communications using the local channels are instantaneous, i.e. the message is enqueued, transmitted and received in the same instant. In practice, the local communications happen inside the ISR. The effect of executing the send statement is not visible outside the ISR except transmitting to an external channel and a computation release. In other words, the only visible actions that result from executing the ISR are sending a message to an external channel and releasing a computation. Based on that, the local communication can be safely assumed to happen either at the beginning of the ISR or at the end of the ISR.

In the following we discuss a problem that may arise when multiple send statements become ready for execution at the same instant.

4.5.2 Multiple Ready Transmissions

Consider the following process fragment:

$$(k!i.x; idle) \mid (k!j.y; idle) \mid (k?i.z; [C_1 : t] + k?j.w; [C_2 : t])$$

Three processes are composed using the parallel operator. The first and the second process send messages with particular identifiers on the channel k. The third process waits to receive a message and execute a particular computation. At some point, the first and the second process may become ready for execution simultaneously. According to the semantics of the send, the message i and j are enqueued at the same instant for transmission. When the processes are allocated into the same node and the channel k is local, the two messages should be sent and received at the same instant because the communication on the local channel is assumed to happen instantaneously. The problem is that if the message i is received first, then the message j can not be received and vice versa because of the choice operator in the third process. In other words, the given model exhibits two possible behaviours at the same time and the choice between them can be made non-deterministically. However the implementation has now a freedom to implement either one of the possible behaviour.

This problem does not arise when the processes are fully distributed on separate nodes and communicate by an external channel. This is because transmission on the external bus consumes a time and the highest priority message will be selected for transmission once the arbitration on the bus begins between the nodes. For instance, if the message i is higher priority than j, then the message i is transmitted first. When the message is successfully received, i.e. $k?i.z$ is executed, the computation $[C_1 : t]$ will then be ready to run and the second term $(k?j.w; [C_2 : t])$ of the third process will be disabled.

Another issue may arise if the first and the second process are allocated to

the same node, then it must be ensured that there are an adequate number of transmit buffers to accommodate all the ready-to-transmit messages. The number of the transmit buffers is a feature of the adopted platform. For example, the Motorola msCAN controller has only three transmit buffers (MC68HC08, 2012). However, it is possible to benefit from the model that is generated from the system description and use the model checking tool to ensure that the implementation never requires more than the available buffers. Therefore, for a particular platform, a design is either accepted or rejected. If the design is rejected, the user can modify the design or use a different platform.

4.6 Process Combinators

The bCANDLE modelling language has a set of control flow operators which are sequential composition, choice, interrupt, and parallel composition. These operators are used to combine the language primitives in order to construct a system model. The code generator derives the system implementation from the net that is generated from the system model. So the implementation implements directly the flow control of the net. This means that the algorithm used to control flow of the execution of the code is exactly the same algorithm that controls the flow of the model. This algorithm is defined by the net rules $R.1$ and $R.2$ discussed in Chapter 2. The rules are used to determine the next marking of the net. Because the code generator is based on the net implementation, the control flow constructs are automatically correct i.e. they behave similarly both in the model and in the generated code. As long as the primitives are implemented correctly and the control flows are implemented correctly, then the implementation behaviour should comply with the model behaviour.

4.7 Summary

In this chapter we have discussed the modelling and the implementation decisions that we have made to ensure that the system model conservatively approximates its implementation. The discussion has been presented for each primitive of the formal language, including the computation, guard evaluation, message reception, and message transmission. Having correctly implemented the primitives and the the control flow of the system model, we conclude that the behaviour of the system implementation complies with the behaviour of the system model.

5. ATOMIC UPDATE OF DATA

As introduced in Chapter 2, a bCANDLE model of a system consists of basic process terms such as: send a message $k!i.x$, receive a message $k?i.x$, perform a computation within a bounded period of time $[\omega : t_1, t_2]$, and evaluate a guard on the data environment γ. These terms may be compounded by a small set of operators: sequential composition ;, choice +, interrupt [>, and parallel composition |. The bCANDLE semantics assumes that computations complete and update their data instantaneously and atomically on completion. In this chapter, we discuss the problem concerning the implementation of computations in which computations are vulnerable to interruption. First, the importance of the interrupt operator in the bCANDLE is outlined in section 5.1. Then, a problem that may arise when implementing a computation with the interrupt operator is discussed in section 5.2. Next, other approaches that resolve the problem are reviewed in section 5.3. After that, a number of methods are proposed to work out the problem in section 5.4. The proposed methods are evaluated in order to select those suitable for the implementation of the code generator in section 5.5. Finally, section 5.6 concludes the chapter.

5.1 The Interrupt Operator

The bCANDLE statement $P[> Q$ behaves as P until either Q performs an action or P terminates. The bCANDLE interrupt operator allows time to pass and network activity to occur in both arguments. It has the same semantics as the interrupt operator of ET-LOTOS (Léonard and Leduc, 1997). The operator

can be used to construct a model where the execution of a process can be disabled when an event occurs (Nicollin and Sifakis, 1994). This allows one to state the responsiveness property of the system when it reacts or responds to environment events (Kesten and Pnueli, 1991). For example, $P[> \beta; Q$ is a process that is allowed to behave as P as long as the termination action of β does not occur. β can be a message reception $k?i.x$, a time delay $[timer : t]$, or guard evaluation γ. If β terminates (e.g: a message is received), then P is aborted and Q begins its execution. The interrupt operator allows a very compact representation of a system behaviour. To illustrate the process expression: $(a_1; a_2; a_3)[> (b_1; b_2)$ may be recast without the interrupt operator. a_3 is the termination action of the first argument, and b_1 is the initial action of the second argument. This can be accomplished using the choice operator as follows:

$$(b_1; b_2) + (a_1; b_1; b_2) + (a_1; a_2; b_1; b_2) + (a_1; a_2; a_3)$$

Fig 5.1 and 5.2 show two nets that are generated from the interrupt expression and the choice expression respectively. It is clearly seen that using the interrupt operator provides a more compact representation than the choice operator in realising the same behaviour.

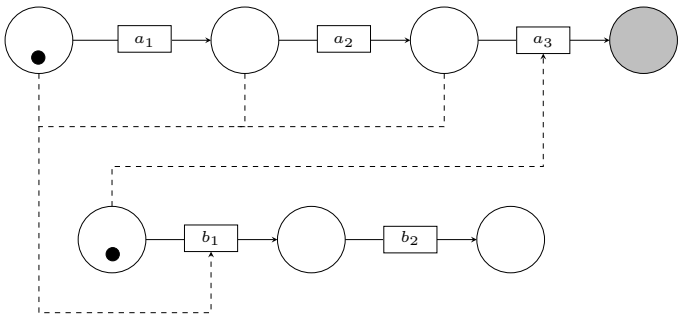


Fig. 5.1: Net of the interrupt expression.

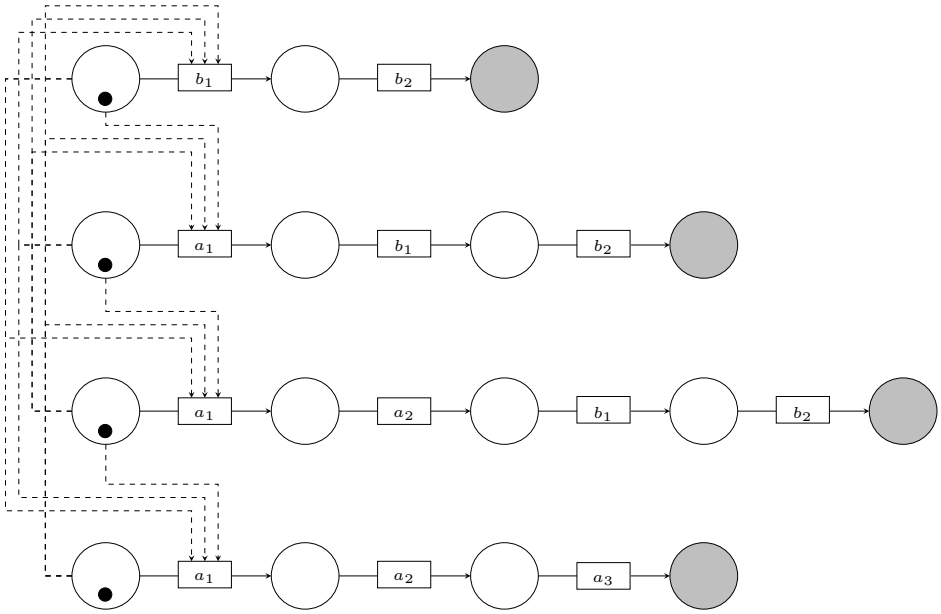


Fig. 5.2: Net of the choice expression.

5.2 The problem of the Interrupt Operator

The bCANDLE semantics assumes that computations complete and update their data instantaneously and atomically on completion. The bounded computation $[\omega : t_1, t_2]$ transforms the data state according to the specification of the operation ω. The change to the data state must occur in a single instantaneous action at the moment of termination. The computation may be compounded with other primitives using the interrupt operator in one of the following three forms:

- $[\omega : t_1, t_2]\ [> [timer : t]; S$ – timeout.
- $[\omega : t_1, t_2]\ [> k?i.x; S$ – message reception.
- $[\omega : t_1, t_2]\ [> \gamma \rightarrow S$ – guard evaluation.

Computations are not allowed to be used as the second argument of the interrupt operator since they can change the data state. In other words, there is no

possibility of interference in the interrupt operator such as in the parallel operator. The computation $[timer : t]$ only consumes time and it does not change the data state, therefore it can be used as second argument for the interrupt operator. For example, assume the following bCANDLE expression:

$$[Comput_1 : t1]; idle[> [timer : t]; [Comput_2 : t2]$$

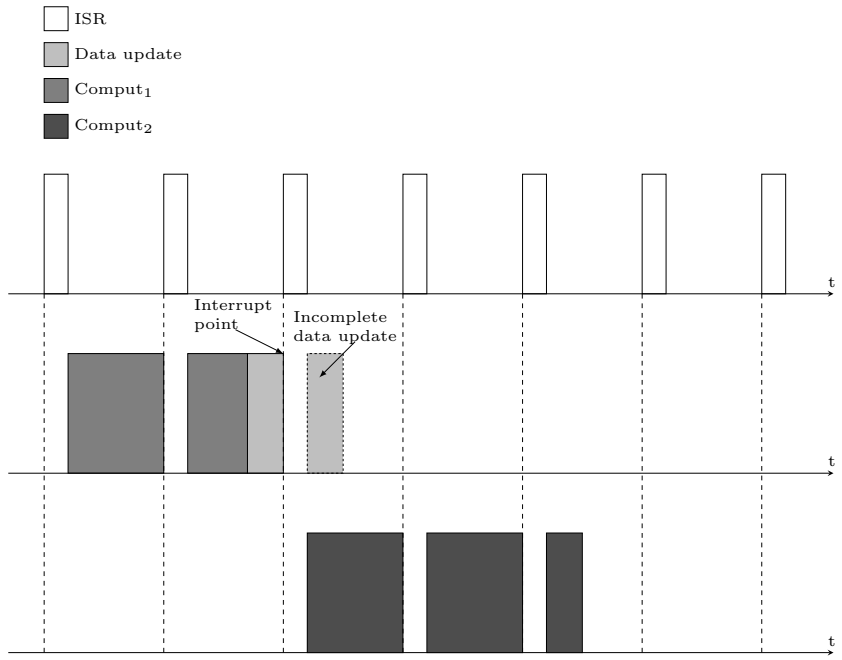


Fig. 5.3: Interrupt problem.

In this expression the computation $[Comput_1 : t1]$ is executed first and time is allowed to progress in the second argument. If the time expires in the time-consuming computation $[timer : t]$, the computation $[Comput_2 : t2]$ starts executing. If the timer expires in $[timer : t]$ and the first argument is in the idle state, then $[Comput_2 : t2]$ can begin executing using the previous data state which will have been updated in $[Comput_1 : t1]$. However, the timer may expire at a point where $[Comput_1 : t1]$ is updating some data state. This could happen when $t_1 > t$. Then $[Comput_1 : t1]$ will be interrupted and the control will be given to $[Comput_2 : t2]$. The data state may be left partially modified because

of the interruption, see Fig. 5.3. Therefore, special attention must be taken to implement the interrupt operator. All partial changes that may have happened to the data state must be aborted at the point of interruption. Consequently, a particular mechanism should be provided during the implementation to ensure that incomplete computation leaves the data identical to the state before start of the computation. This is a well-known problem in pre-emptive multi-tasking systems where a number of tasks may have access to a 'critical section' such as a shared area of memory (read/write global variables) or an input/output ports. In the next section, we review a number of methods and techniques to resolve this problem.

5.3 Related Work

The problem discussed above is a known problem in the area of concurrent systems, and a number of solutions have been proposed. The most common approach is to use a kind of lock, known as a *semaphore*, when accessing a shared resource. A process locks a semaphore and then accesses the resource. When it finishes with the resource, it unlocks the semaphore. Any other process needing access to the shared resource is forced to wait until it can acquire the lock. Using a semaphore requires careful programming practice otherwise it may raise a number of problems. For instance, if one occurrence of a semaphore is omitted or misplaced in a program, the entire program may collapse at run-time (Burns and Wellings, 2001, p. 244). Furthermore, as the semaphore blocks a calling process, a deadlock could occur, if semaphores access is not nested correctly, which in turn may lead to a software failure. Moreover, priority inversion (Mall, 2009) is another problem could happen because of using semaphores. In this case, a high priority process may be blocked for an unbounded time in accessing a shared resource that is locked by a low priority process. In order to avoid the priority inversion problem, priority inheritance protocols were de-

veloped (Goodenough and Sha, 1988; Sha et al., 1990). Many real-time operating system (RTOS) support this, such as VxWorks (WindRiver, 1999), MicroC/OS-II (Labrosse, 2002), and FreeRTOS (Barry, 2009). Additionally, using semaphores is characterised by high memory requirements because the state (context) of each blocked process should be saved in memory. Thus a large number of processes (perhaps more than 100) would lead to unacceptable memory usage (Poledna et al., 1996).

Alternatively, a lock-free approach was proposed by Herlihy and Moss in (Herlihy and Moss, 1993) by using 'transactional memory' (TM). The concept of the transaction first emerged in the area of database management systems. The TM approach is implemented in two ways: hardware TM (HTM) and software TM (STM) approach. The HTM approach relies on hardware support to execute a *transaction* may be used. The transaction is defined as a set of operations that are executed by a process satisfying 'serializability' and 'atomicity' (Herlihy and Moss, 1993). The first term means that the steps of a transaction do not appear to be interleaved with the steps of another transaction, so the transaction seems to execute serially. The second term means that when a transaction completes a sequence of changes to shared places, it commits and then the changes become visible, or aborts and all changes are then discarded. The STM approach was introduced in (Shavit and Touitou, 1997). It provides a software-based implementation of memory transactions exploiting the increase in processors speed. The HTM approach provides better performance but it has architecture limitations in addition to the cost of special purpose hardware for implementation. The STM approach allows larger size of transactions but the implementations suffer from the overhead required to manage transactions (Mankin et al., 2009; Cascaval et al., 2008).

Another approach however deals with the source of the problem rather than providing a solution. In the co-operative scheduling approach of (Pont, 2008b),

the problem does not arise because only one task is active at any time. A task runs to completion and can not be pre-empted by other tasks. As pre-emption is eliminated, the need for a mechanism to protect shared resources does not appear. Considering this approach requires re-design of the system in order to satisfy the assumptions made by the approach such as the duration of tasks must be less than the schedule tick interval (Pont, 2008b).

Additionally, some computer architectures provide some instructions that can read and write memory locations atomically (can not be interrupted by interrupts). For example, in the Motorola HC08, the LDHX instruction can load a 16-bit memory location to the index register (MC68HC08, 2012). Similarly STHX instruction can store the 16-bit index register in a memory location. Furthermore, ARM processors have the Load/Store Multiple (LDM/STM) instruction which can transfer multiple registers of 32-bit size between memory and the processor in a single instruction (Sloss et al., 2004). The main drawback of using these instructions to update data is that they may increase the interrupt latency since they are not interrupted while executing. For example, LDM requires $(2 + Nt)$ cycles to complete execution, where N is the number of registers to load and t is the number of cycles required for each sequential access to memory (Sloss et al., 2004). Although a solution is proposed in (Maaita and Pont, 2005) to reduce the impact of such instructions on the interrupt, this method is a hardware specific and is not appropriate if the data update size is large.

5.4 The Proposed Solution Ensuring Atomic Update

In this section, we discuss a number of methods that ensure the atomicity of data update. For each method, the worst-case response time of a computa-

tion is calculated as the means of comparison. We adopt a traditional analysis approach to calculate the response times. We identify the following three characteristics to compare between the proposed methods:

1. ISR release jitter,
2. worst-case computation completion time,
3. ease of implementation.

Some of the methods could cause a delay to the next invocation of the ISR because of the way they implement the atomic update. This delay is called ISR release jitter. The problem with this delay is that it increases the worst-case execution time of the ISR. The ISR execution time has been included explicitly in our analysis when we produce the lower/upper bound of computations as discussed in Chapter 4. Therefore, the method that minimises the ISR release jitter would yield better analysis (less pessimistic result). The completion time of a computation expresses the time needed to execute the computation and to update its data. Then, the method that yields a shorter worst-case completion time will be considered. The third criterion considers the method that is easier to implement, i.e. the method that does not require more resources to be implemented such as CPU time, memory, or hardware timers.

Additionally, the analysis provided in this work considers only a single process running on a single node, so all computations come from the same net. When a number of processes are allocated to the same node, some computations may have to run at the same time. We have discussed a number of scheduling approaches in Chapter 3. Each proposed method implements different approach to ensure the data update atomicity, so their efficiencies are different in each case. However, when we add the consideration of multi-process to the methods, the additional process affects all methods in the same way and the only changes

occur to the computation times in a consistent way for all methods. These changes do not assist to decide between the methods.

In the following, section 5.4.1 introduces the traditional analysis, and section 5.4.2 proposes the atomic update methods.

5.4.1 Worst-Case Response Analysis

The exact analysis of Joseph and Pandya (Joseph and Pandya, 1986) calculated the worst-case execution time (response time) of a task in a system with pre-emptive task scheduler. The system behaviour is limited to the following computation model in order to do this kind of analysis. All tasks have periods. All deadlines are equal to these periods. All tasks are independent and do not communicate with each other. Finally, a task has a unique priority level according to the rate monotonic policy in which the shorter the period, the higher the priority. The equation (Joseph and Pandya, 1986) below shows how the worst-case response time can be computed iteratively:

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{5.1}$$

Here are the definition of the parameters in Equation (5.1):

- C_i: worst-case computation time of task i.
- T_i: period of task i.
- R_i: worst-case response time of task i.
- n: the current iteration number.
- $hp(i)$: a set of tasks of higher priority than task i. The

Response Time Analysis with Release Jitter

Equation (5.1) assumes that there is no delay between the invocation time - arrival time - of a task and release time - actual running time - of the task. The strict periodicity assumption now is relaxed. The following equation takes into account the variable delay between the invocation and release time of a task:

$$\begin{aligned} w_i^{n+1} &= C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n + J_j}{T_j} \right\rceil C_j \\ R_i &= w_i + J_i \end{aligned} \tag{5.2}$$

Here are the definition of the parameters in the equation:

- w_i: worst-case response time of task i once it has been released.
- R_i: worst-case response time of task i including the delay between invocation and release.
- n: the current iteration number.
- J_j: is release jitter of a higher priority task j, the difference between the longest and shortest delay from invocation to release of the task.

5.4.2 Atomic Update Methods

In order to apply the traditional response time analysis on the atomic update methods, we employ the following assumptions. First, the ISR is treated as a periodic task that runs with the highest priority level. The period of the ISR is denoted by T, and the worst case computation time of the ISR is expressed by Cs. Second, computations are considered to execute in the lower priority level. A computation has a worst-case computation time expressed by C_i, and the worst-case response time is denoted by R_i. The worst-case time required by

a computation i to update its data state is represented by A_i. The completion time of the computation is expressed by r_i. Third, all ready computations are assumed to be released at the beginning of the ISR. This assumption complies with the *critical point* assumption of the traditional analysis at which all system components (tasks) are assumed to be released at the same time. Forth, the duration of the atomic update of a computation must be accommodated within one tick interval: $A_i < (T - Cs)$. This is because a longer update operation would lead to missing one tick and the track of the elapsed time as a consequence. However, a long atomic update of data could be split into a number of short operations by the programmer in a way that satisfies the previous constraint.

Then, for each method, we calculate the worst-case response time of a computation in the basic case that does not take into account the effect that a method has on the ISR release jitter or the time of the atomic update of data. Next, we calculate the computation response time by taking into account only the ISR release jitter when presented. After that, we calculate the worst-case completion time of the computation which includes the basic response time and the atomic update time.

Method(1): One-tick duration computation

Similarly to the TT approach (Pont, 2008b), when a computation completes in one-tick time, then the data is guaranteed to be modified without interference of other computations. Fig. 5.4 shows two short sequential computations. The computation 1 and 2 are completed within one tick interval. When the first computation updates data before completion, then the update operation performed by the computation 1 is ensured to be completed before running the computation 2. Consequently, the main assumption in this approach is that the worst-case response time of a computation plus the data update time must

be less than the tick interval time: $R_i + A_i < T$.

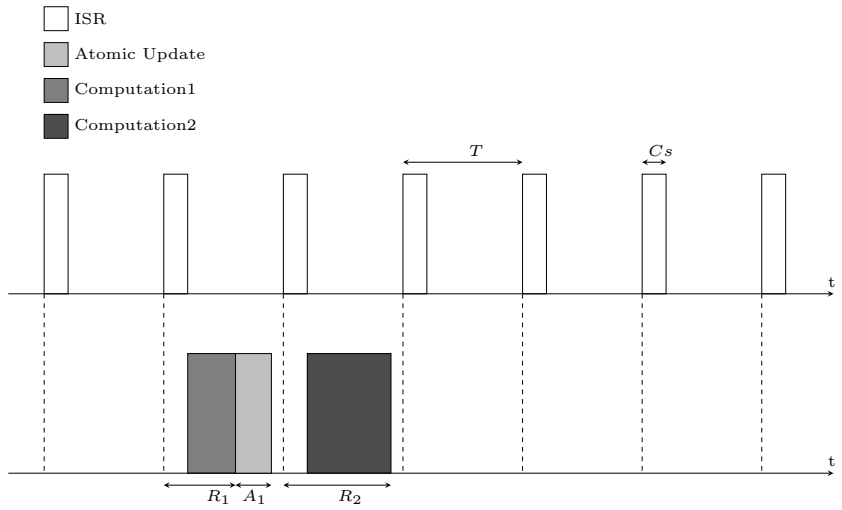


Fig. 5.4: Method(1): One-tick duration computation.

The worst-case response time of the computation is simply calculated by Equation 5.3:

$$R_i = C_i + Cs \tag{5.3}$$

This method has no impact on the ISR release time because a computation completes within one tick time.

The worst-case completion time of a computation is calculated by Equation 5.4:

$$r_i = R_i + A_i \tag{5.4}$$

Method(2): Atomic Update with Enable/Disable Interrupts

A computation may disable the timer interrupt before updating the data state. Once the computation completes, the data is modified and the timer interrupt is enabled thereafter. The advantage of this solution is that it introduces no delay between the computation and the data update. However, the update operation may exceed the time available in the current time slot because of a large size of data that needs to be changed. This will cause a release jitter to

the next invocation of the ISR (represented by J_s) before it is actually started, see Fig. 5.5 for example. In the worst-case, the ISR release jitter could equal A_i. This may happen when the computation starts data update operation just before the beginning of the next invocation of the ISR. Additionally, the method provides no opportunity to interrupt the current-running computation once it starts an update operation.

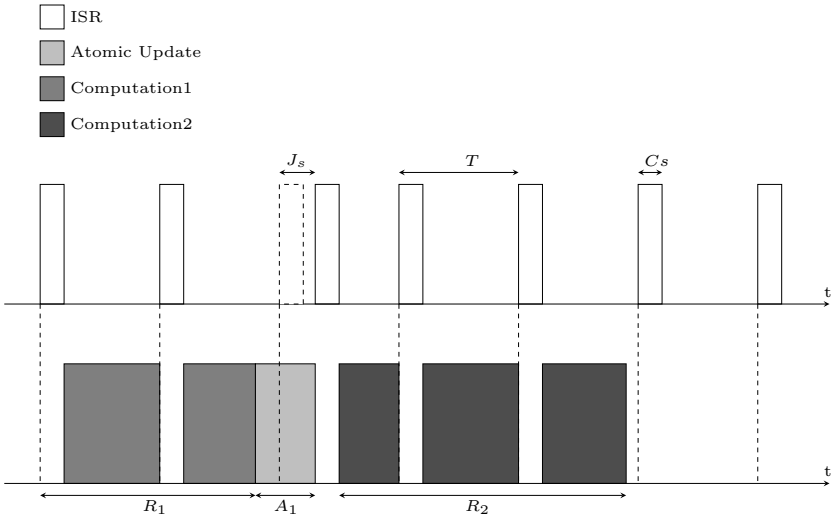


Fig. 5.5: Method(2): Computation disables interrupts.

The worst-case response time of a computation is calculated by the equation 5.5.

$$\begin{aligned} R_i^{n+1} &= C_i + \left\lceil \frac{R_i^n}{T} \right\rceil Cs \\ R_i^0 &= C_i \end{aligned} \tag{5.5}$$

The worst-case response time of a computation that suffers from an ISR release jitter is calculated by the equation 5.6:

$$\begin{aligned} R_i^{n+1} &= C_i + \left\lceil \frac{R_i^n + J_s}{T} \right\rceil Cs \\ R_i^0 &= C_i \\ 0 &\leq J_s \leq A_{i-1} \end{aligned} \tag{5.6}$$

where A_{i-1} is the atomic update time needed by a preceding running computation.

The worst-case completion time of a computation is calculated by the equation 5.7:

$$r_i = R_i + A_i \tag{5.7}$$

Method(3): Atomic Update Inside the Interrupt Handler

In this method, a computation does not update the data state. The computation completes and then idles. The data is updated within the next execution of the ISR. Although this method ensures the strict periodicity of the tick timer, it introduces a delay to the execution of a succeeding ready-to-run computation. For example, in Fig. 5.6, computation 1 idles for d_1 time unit before the data is updated in the next invocation of the ISR. Furthermore, a large size of data may cause a large overhead to the ISR execution time, leaving no time to process a new ready computation in the next time slot. Similarly to the previous method, this method introduces an ISR release jitter which equals the time of the atomic update.

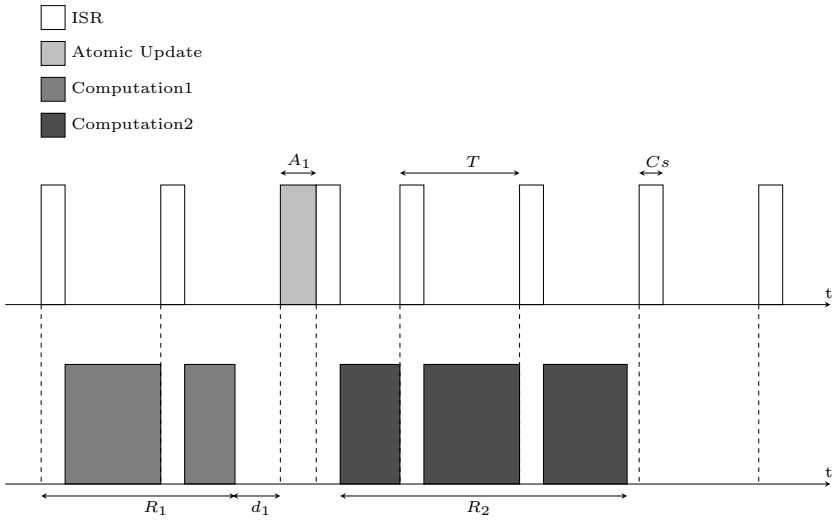


Fig. 5.6: Method(3): Update data inside the interrupt handler.

The worst-case response time of a computation is calculated by the equation 5.8.

$$\begin{aligned} R_i^{n+1} &= C_i + \left\lceil \frac{R_i^n}{T} \right\rceil Cs \\ R_i^0 &= C_i \end{aligned} \tag{5.8}$$

The worst-case response time of a computation that suffers from an ISR release jitter is calculated by the equation 5.9:

$$\begin{aligned} R_i^{n+1} &= C_i + \left\lceil \frac{R_i^n + J_s}{T} \right\rceil Cs \\ R_i^0 &= C_i \\ J_s &= A_{i-1} \end{aligned} \tag{5.9}$$

where A_{i-1} is the atomic update time needed by a preceding running computation.

The worst-case completion time of a computation is calculated by the equation 5.10:

$$\begin{aligned} r_i &= R_i + d_i + A_i \\ d_i &= \left\lceil \frac{R_i}{T} \right\rceil T - R_i \end{aligned} \tag{5.10}$$

Method(4): Atomic Update with Rollback

In this method, a computation is allowed to modify its data state immediately after it has completed. A roll-back mechanism is provided to guarantee that the interrupted update operation leaves the data identical to the state before the execution of the operation. The proposed roll-back mechanism is illustrated in Fig. 5.7. A computation first obtains a local copy of the data state. When the computation is completed, it saves a copy of the original data sate in a *log* and then updates the global data state. The roll-back mechanism is important to protect the data from partial changes because of the interrupted (or incomplete)

update operation, and to restore the data to the previous consistent state. However, a considerable interrupt overhead may occur if the update operation is interrupted at the point just before it terminates. In this case, a full recovery procedure must be performed to all changed data, which would introduce a delay to a succeeding ready-to-run computation. For example, in Fig. 5.8, the update operation of computation 1 is aborted in the next invocation of the ISR before it terminates. In this case, a roll-back recovery operation is undertaken to restore the original state of the data. It consumes B_1 time units before the new ready computation 2 is dispatched. Similarly to the previous method, a computation may suffer from the ISR release jitter.

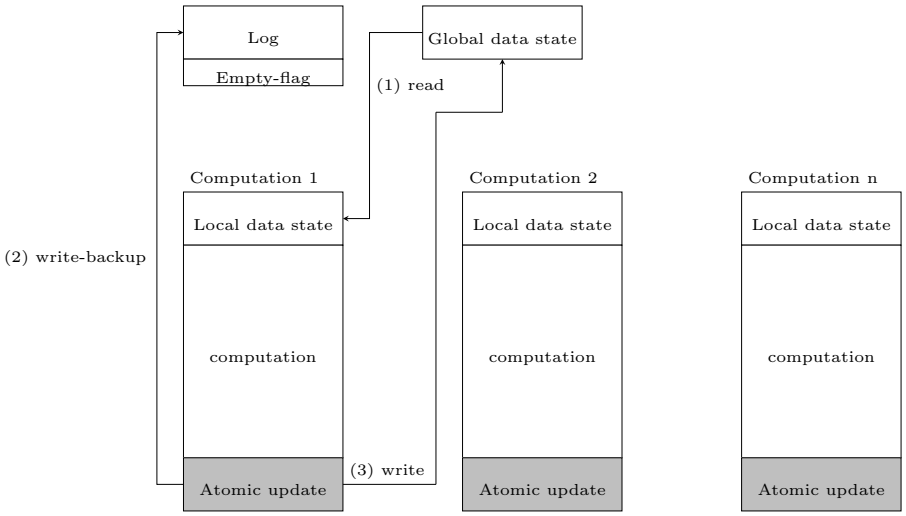


Fig. 5.7: Roll-back mechanism.

The worst-case response time of a computation is calculated by the equation 5.11.

$$\begin{aligned} R_i^{n+1} &= C_i + \left\lceil \frac{R_i^n}{T} \right\rceil Cs \\ R_i^0 &= C_i \end{aligned} \tag{5.11}$$

The worst-case response time of a computation that suffers from an ISR release

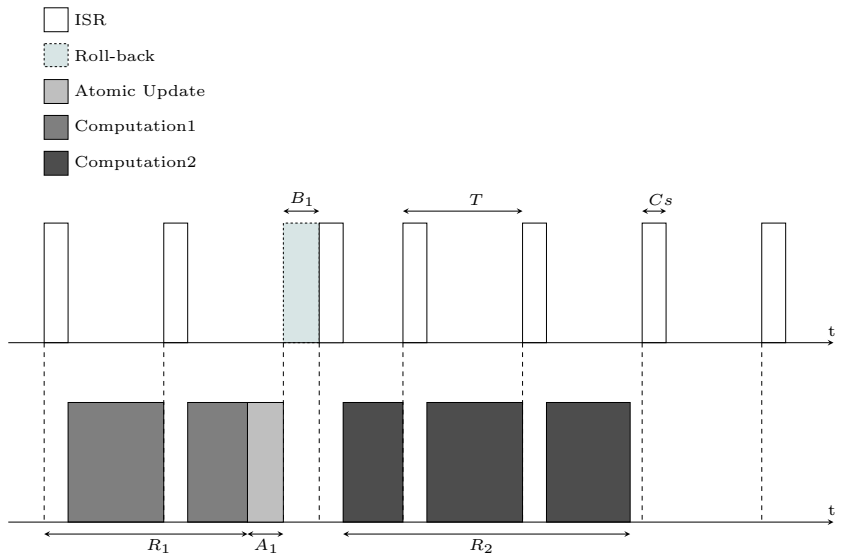


Fig. 5.8: Method(4): Update data with roll-back.

jitter is calculated by the equation 5.12:

$$\begin{aligned} R_i^{n+1} &= C_i + \left\lceil \frac{R_i^n + J_s}{T} \right\rceil Cs \\ R_i^0 &= C_i \\ J_s &= B_{i-1} \end{aligned} \tag{5.12}$$

where B_{i-1} is the time needed to do a roll-back recovery of data of a preceding running computation. Similarly to the constraint we adopt for the time of the atomic update, it must be ensured that the recovery time completes in the worst-case within one tick interval (i.e, $B_{i-1} < T - Cs$).

The worst-case completion time of a computation is calculated by the equation 5.13:

$$r_i = R_i + A_i \tag{5.13}$$

Method(5): Delayed Atomic Update

This method always assumes that there is insufficient time to update the data state of a computation in the current time slot, see Fig. 5.9. When the computa-

tion is completed, it postpones the data update operation to the next available time slot. The main advantage of this method is that the ISR has the opportunity to abort the current running computation before any changes may occur to the data state, and run a new computation after that.

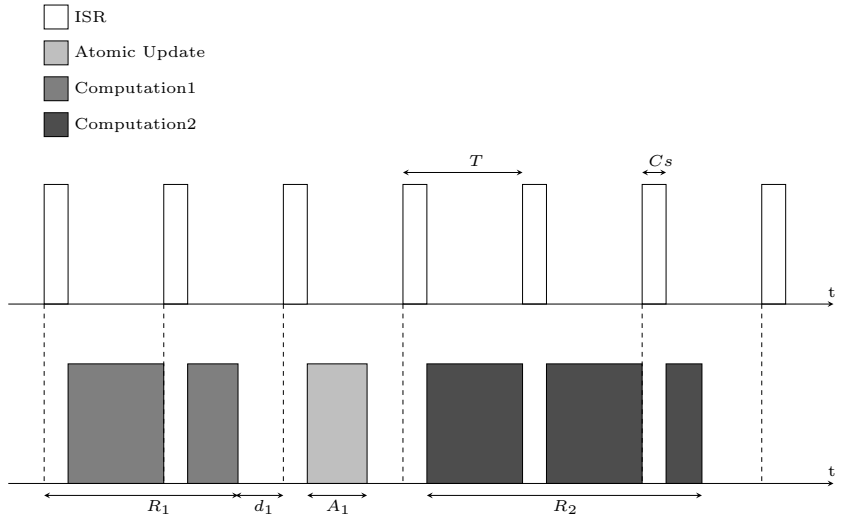


Fig. 5.9: Method(5): Delayed atomic update.

The worst-case response time of a computation is calculated by the equation 5.14.

$$\begin{aligned} R_i^{n+1} &= C_i + \left\lceil \frac{R_i^n}{T} \right\rceil Cs \\ R_i^0 &= C_i \end{aligned} \tag{5.14}$$

This method has no impact on the ISR release time.

The worst-case completion time of a computation is calculated by the equation 5.15:

$$\begin{aligned} r_i &= R_i + d_i + Cs + A_i \\ d_i &= \left\lceil \frac{R_i}{T} \right\rceil T - R_i \end{aligned} \tag{5.15}$$

Method(6): Update Data Now or Delay

A computation checks the remaining time in the current time slot to see if there is a sufficient time to update the data state before the next invocation of the ISR. If this is the case, the computation immediately changes the data in the current slot, otherwise the update operation is delayed to the next available time slot and the computation idles. Fig. 5.10 illustrates the two cases. In the figure, computation 1 modifies the date immediately when it is completed since the remaining time d_1 is longer than the time needed to update the data A_1. Computation 2, however, delays the update operation because the remaining time d_2 in the current slot is shorter than the time needed to complete the update operation A_2. The advantage of this method is that the ISR has always the opportunity to cancel the current running computation before any changes may occur to the data state, and to run a new computation after that.

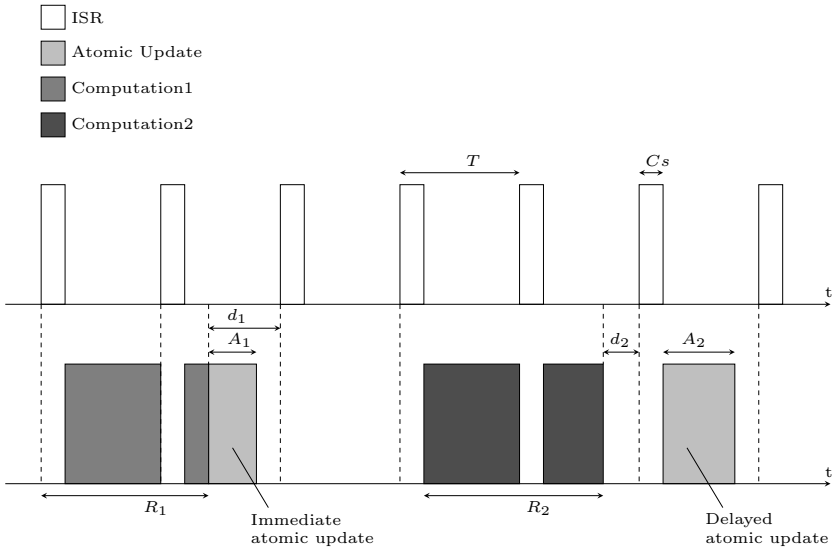


Fig. 5.10: Method(6): Update now or delay update.

The worst-case response time of a computation is calculated by the equa-

tion 5.16.

$$\begin{aligned} R_i^{n+1} &= C_i + \left\lceil \frac{R_i^n}{T} \right\rceil Cs \\ R_i^0 &= C_i \end{aligned} \tag{5.16}$$

This method has no impact on the ISR release time.

The worst-case completion time of a computation is calculated by the equation 5.17:

$$r_i = \begin{cases} R_i + A_i & \text{if } A_i \leq d_i \\ R_i + d_i + Cs + A_i & \text{if } A_i > d_i \end{cases} \tag{5.17}$$

$$d_i = \left\lceil \frac{R_i}{T} \right\rceil T - R_i$$

Method	Worst-case response time	Worst-case response time with ISR release jitter	Worst-case completion time
1	$R_i^{n+1} = C_i + Cs$	*none*	$R_i + A_i$
2	$R_i^{n+1} = C_i + \left\lceil \frac{R_i^n}{T} \right\rceil Cs$	$R_i^{n+1} = C_i + \left\lceil \frac{R_i^n + J_s}{T} \right\rceil Cs$	$R_i + A_i$
3	$R_i^{n+1} = C_i + \left\lceil \frac{R_i^n}{T} \right\rceil Cs$	$R_i^{n+1} = C_i + \left\lceil \frac{R_i^n + J_s}{T} \right\rceil Cs$ $J_s = A_{i-1}$	$R_i + d_i + A_i$ $d_i = \left\lceil \frac{R_i}{T} \right\rceil T - R_i$
4	$R_i^{n+1} = C_i + \left\lceil \frac{R_i^n}{T} \right\rceil Cs$	$R_i^{n+1} = C_i + \left\lceil \frac{R_i^n + J_s}{T} \right\rceil Cs$ $J_s = B_{i-1}$	$R_i + A_i$
5	$R_i^{n+1} = C_i + \left\lceil \frac{R_i^n}{T} \right\rceil Cs$	*none*	$R_i + d_i + Cs + A_i$ $d_i = \left\lceil \frac{R_i}{T} \right\rceil T - R_i$
6	$R_i^{n+1} = C_i + \left\lceil \frac{R_i^n}{T} \right\rceil Cs$	*none*	$R_i + A_i$, **if** $A_i \leq d_i$. **Or** $R_i + d_i + Cs + A_i$, **if** $A_i > d_i$ $d_i = \left\lceil \frac{R_i}{T} \right\rceil T - R_i$

Tab. 5.1: Response times and completion times of atomic update methods.

5.5 Evaluation and Discussion

Six methods have been proposed to resolve the problem of atomic update. We calculate, in each method, the worst-case response time of a computation in the basic case, the worst-case response time of a computation that suffers from the ISR release jitter (when presented), and the worst-case completion time of a computation. The results are summarised in Table 5.1. The response times of a computation are computed similarly in all methods, see the first column of the table. The response time of the method 1 can be considered a special case of the response time of the remaining methods because computations, according to this method, complete within one tick interval and so the term $\left\lceil \frac{R_i^n}{T} \right\rceil$ equals 1. In the second column, the response time of a computation is calculated when the computation experiences ISR release jitter. The ISR release jitter is caused by the overrun of a preceding computation and it is related to the data update method. The methods 2, 3, and 4 introduce ISR release jitter and so they have impact on a succeeding running computation. The methods 1, 5, and 6, however, do not introduce ISR release jitter. The third column shows the calculation of the completion time of a computation which includes the response time of the computation and the data update time. The completion time varies according to the methods used to update data. The methods 1, 2, and 4 have similar way of calculation of the completion time which is the summation of the response time and data update time only. The methods 3 and 5 require calculating d_i, the idle (waiting) time that the computation has to wait until the update operation is started, in addition to the response time and data update time.

Based on the first criterion introduced in section 5.4, the methods 2, 3, and 4 are discarded because they introduce ISR release jitter and so may lead to a pessimistic analysis. The methods 1, 5, and 6 pass this criterion because they have no impact on the ISR release jitter.

When the system computations are short and can complete within one tick time, then method 1 would be suitable for implementation of our code generation approach. Method 1 has very restrictive assumption on the duration of the computations which would be difficult to satisfy if we require long computations since this would require increasing the tick interval and reducing responsiveness of the system. However, this method does not require a special mechanism to ensure the atomicity of the data update. Moreover, a long computation can be split by the programmer into a number of short computations in a way that ensure all computations and their atomic updates complete within one tick interval.

The methods 5 and 6 can be considered when the system computations (or at least one computation) require more than one tick interval to complete. Although, method 6 would give better performance than method 5 during run-time because the data is updated soon after the computation completes, both of them have the same worst-case completion time. Now, according to the third criterion, method 5 is easier to implement than method 6 because method 6 requires access to a hardware timer during run-time in order to evaluate the remaining time before the next invocation of the ISR. Therefore, method 5 would be more suitable for our code generation approach.

A particular mechanism to ensure the atomicity of the data update is not required in the following two cases. First, a computation takes more than one tick interval to complete but it is not affected by the interrupt operator (i.e. not interrupted at all). Second, a computation could take more than one tick interval and the data update is interrupted but the computation does not use the data again before they are reinitialised. These cases can be checked using model-checking. It would require constructing a suitable property to verify that the computation never be affected by the interrupt or makes use of data before reinitialisation. When there is a computation that runs for a number of ticks

and can be affected with the interrupt, then one of the recommended mechanism must be implemented and applied in order to ensure the atomicity of the data update. This however is outside the scope of this thesis. Future work on the atomic update work would require implementing a chosen solution for the code generation, and testing it in a case study.

5.6 Summary

The bCANDLE semantics requires that computations complete and update their data atomically. The syntactic restrictions imposed by CANDLE ensure that the interrupt operator is the only source of potential failure of this requirement. This chapter has examined the problem of the interrupt operator. Several methods have been proposed to resolve the problem, namely one-tick duration computation, atomic update with enable/disable interrupts, atomic update inside the interrupt, atomic update with rollback, delayed atomic update, and update data now or delay. Many of these methods are well-known in the in the context of database transactions (Haerder and Reuter, 1983). We have presented a new analysis in the context of embedded systems by applying response time analysis to calculate the worst-case response time of each method and use this as our basis for comparison.

6. EVALUATION AND EXPERIMENTS

6.1 Introduction

The purpose of this chapter is to demonstrate the feasibility of our code generation approach on a number of case studies. Additionally, we are attempting to highlight the limitations of the approach and assess the complexity of systems that may be analysed with available computing resource. The measures used to demonstrate the practical applicability are the performance and the formal verification capability of the proposed approach. The performance is measured in terms of the computational effort required to generate an executable code and a formal model for a given design, and the computational resources including memory (RAM and ROM) and CPU load required to execute the examples on the target. The experimental results are compared with results taking an alternative approach employing a real-time operating system (e.g, MicroC OS-II) to implement the same examples. To evaluate the formal verification capability of our approach, we generate a formal model (Timed Automata model) for each case study and use a model checking tool (e.g, UPPAAL Model Checker) to check if the system satisfies a number of useful properties including functional and timing properties. The main point of this is to discover to what extend the models which are generated are tractable. Moreover, the computation resources (CPU time and memory) used by the model checker are measured for the properties. This illustrates the cost of the computation resources required to verify a number of different case studies.

The chapter is organised as follows. Section 6.2 briefly presents the main char-

acteristics of the four examples we choose as case studies, namely a flow regulator system, steam boiler control system, security alarm system, and anti-lock braking and anti-slip regulation system (ABS/ASR). Section 6.3 presents our results from the performance evaluation of the code generated for the selected case studies. Section 6.4 provides experimental results that demonstrate the formal verification capability of our approach. Finally, we conclude our results in section 6.5.

6.2 Case Studies

For our practical evaluation, we use four case studies: a flow regulator system, steam boiler control system, security alarm system, and an ABS/ASR system. The flow regulator system is a well-known example in the literature (Kopetz, 1997). Additionally, it is especially well-suited to assess the scalability of our approach because we readily can vary the complexity of the designs. The steam boiler control example has multiple operational modes meaning that the system behaves in different phases during running. This kind of system is very difficult to analyse by traditional methods applicable only to periodic systems. The security alarm example illustrates the usefulness of the local communication mechanism adopted by our approach because all system components are executed on a single-processor platform and communicate only by locally passing messages. The ABS/ASR system was motivated by the fact that it is an industry-related example, and it is recently used in the literature, for example (Enoiu et al., 2012; Herber, 2010).

6.2.1 Flow Regulator System

The purpose of the system is to control the flow rate of a liquid through a pipe according to a set value. The physical view of the system is depicted in Fig. 6.1.

The system consists of two processes that run on separate nodes communicating via a CAN bus. The first process is *Flow* which periodically reads the rate of the flow using a flow sensor and then broadcasts the flow value through the CAN bus. The second one is *Valve* which controls the position of a valve based on the received flow value so that the flow rate remains within a small range around a pre-determined flow value. The system architecture is illustrated in Fig. 6.2. The CANDLE program of the system is shown in Appendix A.

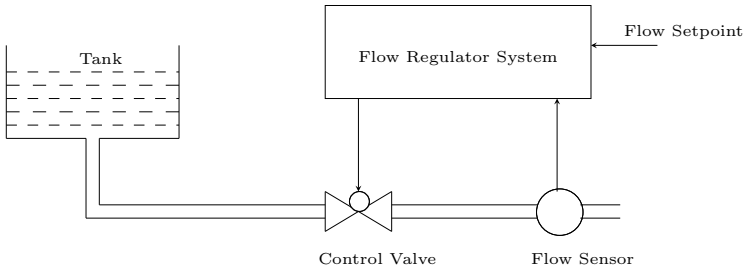


Fig. 6.1: Flow Regulator System.

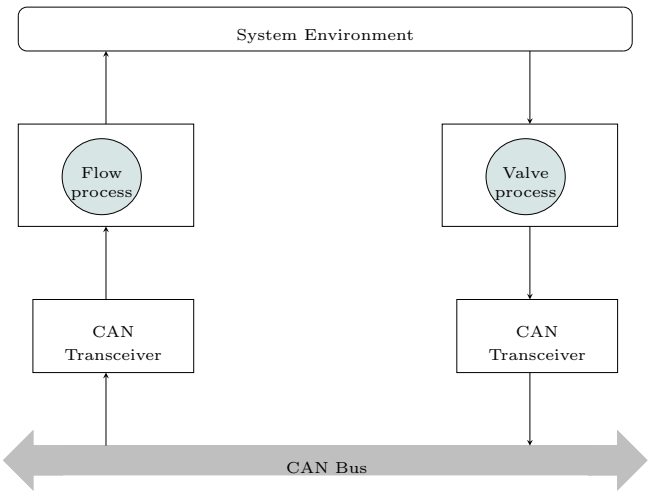


Fig. 6.2: Architecture of the flow regulator system.

In order to evaluate the scalability of our approach, the size of the system design is varied by adding more processes of *Flow* and *Valve*, and then the performance of the generated code for the new designs is evaluated as provided later in section 6.3. The system design is varied according to the following scenarios. We preserve the number of nodes and duplicate the number of processes executing

on each node of the original design. The processes that are allocated to one node communicate with the corresponding processes of the other node using one CAN bus, see Fig 6.3. For the purpose of performance evaluation, the process of each node is duplicated up to three times. So we obtain another two versions of the example called *Flow2* and *Flow3*. Now, Flow2 consists of 2 processes of Flow and 2 processes of Valve, and Flow3 consists of 3 processes of Flow and 3 processes of Valve. In the rest of this chapter, we use Flow1 to refer to the original flow example which consists of 1 Flow process and 1 Valve process.

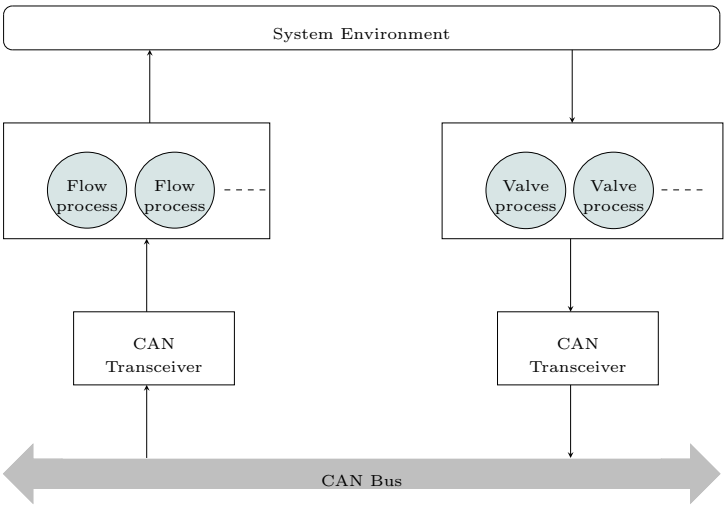


Fig. 6.3: Architecture of the modified flow regulator system.

6.2.2 Steam Boiler Control System

This example is a modified version of the steam boiler control problem described in (Abrial et al., 1996). The system consists of a steam boiler, a pump, and water-level sensor. The pump controls the flow of water into the boiler, which is then heated and evaporated to produce steam. The steam flows out from the top of the boiler and is used to power a generator. The water-level sensor provides the control system with the level of water in the boiler. Fig 6.4 shows a physical view of the system. The objective of the system is to maintain the level of water (w) in the boiler within minimum (W_1) and maximum (W_2) bounds.

This can be performed by turning the pump on or off, so it is ensured that $W_1 \leq w \leq W_2$ is satisfied. It is considered unsafe to operate the system if the sensor process fails. The system detects a sensor process failure when the interval between sensor messages exceeds some threshold.

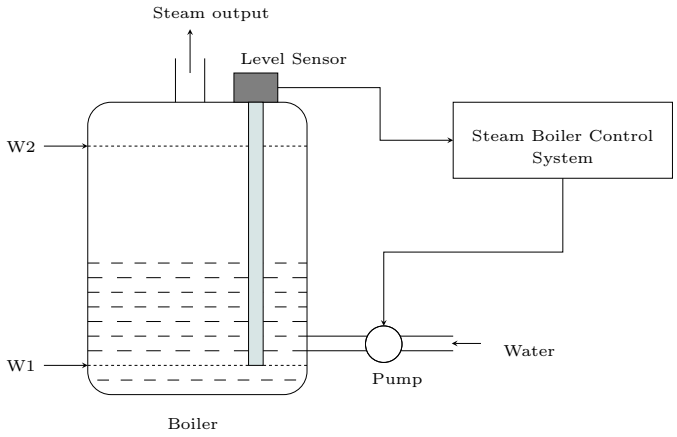


Fig. 6.4: Steam Boiler Control System.

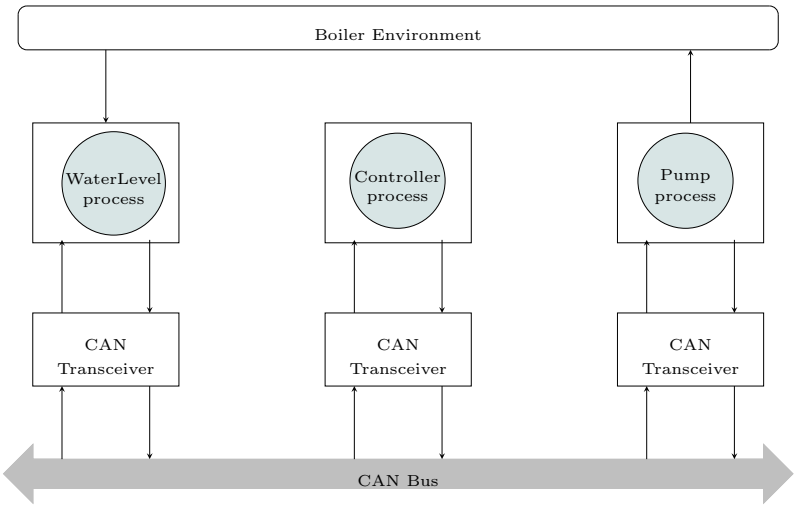


Fig. 6.5: Architecture of the steam boiler control system.

The control system consists of three processes: *Controller*, *WaterLevel*, and *Pump*. Each process runs on a separate node and communicates with other processes via a CAN bus. Fig. 6.5 shows the architecture of the system. The CANDLE program of the system is shown in Appendix A. The system operates in three different modes: *initialisation*, *ready*, and *normal*.

- In the initialisation mode, the system resets its devices and initialises its local data. It is assumed that the system starts with the water level in the boiler between W_1 and W_2 and the pump is off. The system moves to the *ready* mode after successful initialisation.

- In the ready mode, the processes $WaterLevel$ and $Pump$ repeatedly report a *ready* message to the $Controller$ process until they receive a *start* message. After that, the system moves to the *normal* mode.

- In the normal operation mode, the $WaterLevel$ process periodically reads the water-level sensor and broadcasts the current sensor value. The $Controller$ process receives the sensor value and then evaluates the value of the water level. If the level is too high, a message is sent to turn off the pump. If the level is too low, a message is sent to turn on the pump. The pump is kept in the current state if water is within the acceptable level. However, the $Controller$ process sends a *shutdown* message to other processes if it does not receive a sensor level value before timing out and then the system idles. In this case, it is assumed that the water-level sensor is faulty.

6.2.3 Security Alarm System

The security alarm system is a simple theft-detection device (e.g, a briefcase). The system functions as follows. A user presses a button to enable the security mechanism. When the system detects any motion, it requires the user to enter a security code within a pre-defined time interval. The alarm timer starts when the motion is first detected. If the correct code is entered in time, the security mechanism is disabled and the device can be opened by pressing a button. When the correct code is not entered on time, the alarm sounds. However, the alarm can be turned off only by entering the correct security code, after that the security mechanism is disabled. The device can be locked

and unlocked freely if the security mechanism is not enabled. The physical view of the system is depicted in Fig 6.6. In this figure, the button E of the keypad is used to enable/disable the security alarm system, and the button L is used to lock/unlock the device. The LCD is used to display the status of the system.

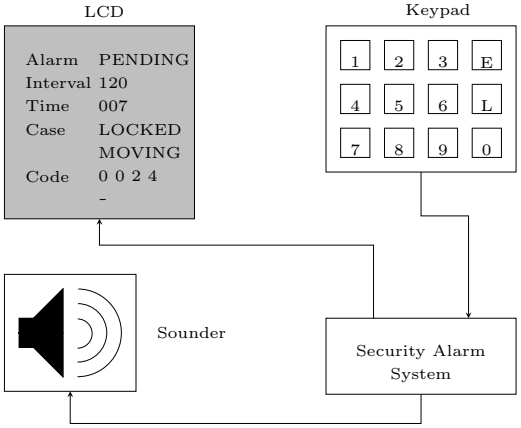


Fig. 6.6: Security Alarm System.

The system comprises four processes that run on the same node and communicate via a local broadcast channel. Fig.6.7 illustrates the system architecture. The main process is *Control* which continuously monitors and reports the status of the device. The *PendingTimer* process represents the alarm timer which keeps track of the current time elapse. The *Flasher* process enables a suitable alert device (e.g, sounder). The *Display* process outputs the current system status to a suitable display device (e.g, LCD). The system processes communicate by exchanging messages using a single local broadcast channel. The CANDLE program of the system is shown in Appendix A.

6.2.4 ABS/ASR System

In a vehicle with a conventional braking system, when the driver depresses the brake pedal in the case of emergency situation, the vehicle wheels may get locked. The locked wheels prevent the driver from steering the vehicle while stopping. This could threaten the driver's life or cause severe damage. To avoid

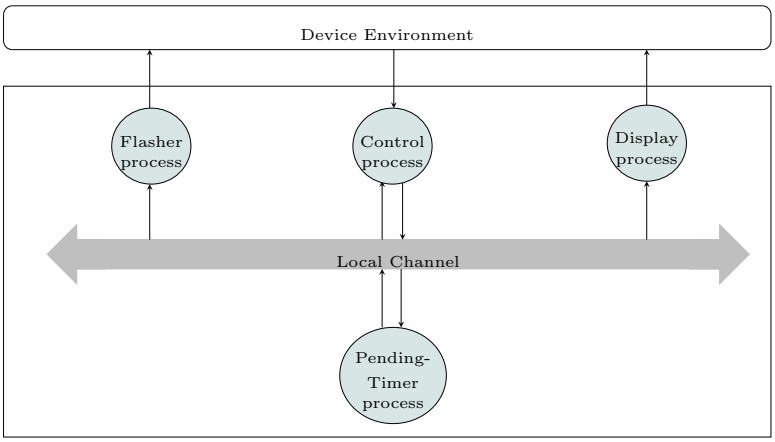


Fig. 6.7: Architecture of the security alarm system.

wheel lockup or loss of traction, the ABS/ASR system (Day and Roberts, 2002; Bosch GmbH, 2011) has been developed . The ABS/ASR system monitors the speed of each wheel and regulates the brake pressure in order to avoid locking a wheel. This improves the driver's control over the vehicle while the brakes are applied.

The system consists of three process types: a *Sensor* process that reports a wheel speed, a *Brake* process that regulates the brake pressure of a wheel, and a *Control* process that executes the ABS/ASR control algorithm. There are four sensor processes and four brake processes in the system. The sensor and brake process are allocated to the same node, whereas the control process is executed on a single node. All processes communicate using a CAN bus. The control process periodically sends a speed request to the sensor processes. The sensor processes send the speed of the wheel to the control process. The control process executes the ABS/ASR control algorithm, and then broadcasts the new pressure values. The brake process of each wheel reads its corresponding pressure value and adjusts the brake pressure accordingly. The physical view of the system and the architecture of the design are shown in Figure 6.8. The CANDLE program of the ABS is shown in Appendix A.

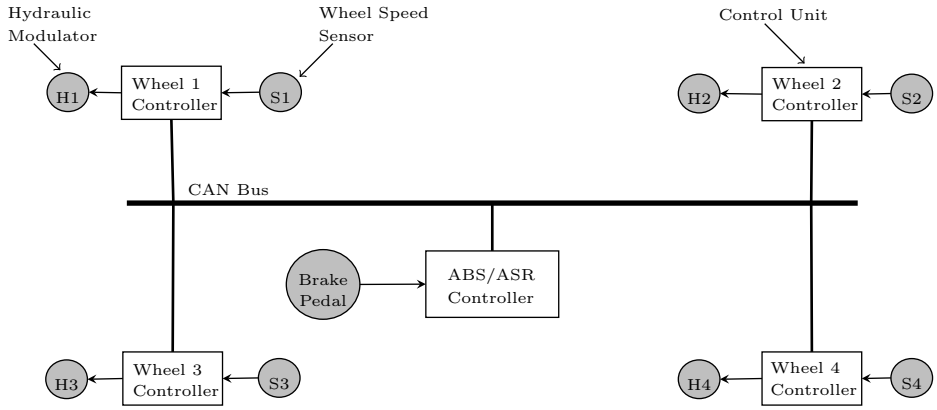


Fig. 6.8: ABS/ASR Control System.

6.3 Performance Evaluation

For the evaluation of the performance of our approach, we measured the computational effort of

- the generation of executable C code for the CANDLE program,
- the transformation of a given CANDLE program into an UPPAAL model.

The code generator program and the model generator program were run on a PC with an AMD Athlon(tm) 3GHz Dual Core Processor 5200B and 2 GB of main memory running a Linux operating system. Table 6.1 shows the main features of the system design for each case study. In this table, *Process* is the number of processes in the CANDLE design of the example, *LOC* is the number of lines of CANDLE code, and *Transition* is the number of transitions of the net generated from the example design. The results show that the examples have small nets (expressed by the number of transitions), e.g the Alarm and ABS example have only a few tens of transitions.

The results of the code and model transformation times are presented in Table 6.2. In this table, *Code-time* represents the time required to generate an executable C code for the example, and *Model-time* expresses the time required

Case-study	Processes	LOC	Transitions
Flow1	2	15	7
Flow2	4	27	12
Flow3	6	39	17
Boiler	3	60	39
Alarm	4	61	32
ABS	9	75	38

Tab. 6.1: Features of the case studies.

to generate an UPPAAL model for the example. The transformation times are given in seconds. The code and model generator program has been written in the Ocaml and Python programming languages using the StringTemplate library (Parr, 2013). The experiment shows that the measured times are acceptable (i.e. performed within a few seconds), and vary a little as a function of the size of the example (number of processes, LOC, and transitions).

Case-study	Code-time (s)	Model-time (s)
Flow1	2.00	3.81
Flow2	2.02	3.83
Flow3	2.06	3.97
Boiler	2.34	4.65
Alarm	2.29	4.77
ABS	2.26	5.18

Tab. 6.2: Transformation times from CANDLE into C and UPPAAL.

For the evaluation of the computational resource of the approach, the memory usage of the executable C code has been measured for each case study. The case studies were implemented using an Olimex LPC-2378STK development prototype board with MCU LPC2378 16/32 bit ARM7TDMI-S^{TM}t processor, 512K Bytes of Program Flash, and 16K Bytes of RAM. The source codes of the examples were compiled using the IAR Embedded Workbench IDE version 6.21 with the compiler setting shown in Table 6.3. The results of the experiments are shown in Table 6.4. When the example is implemented using more than one node, the memory usage is presented separately for each computing node. For instance, the *Flow1* case study consists of two computing nodes: *node_flow* and *node_valve*. The scheduling algorithm employed in each node of the case study is

shown. Currently, we use two scheduling methods: cooperative (co) and round-robin (rr) scheduling method. The memory usage of the application code (which is generated code by our framework) and the device-driver code (which is used to access the IO, e.g. CAN, LCD, LEDs, and ADC) were shown separately, see Table 6.4. For the application code, the RAM and ROM requirements were measured to assess the efficiency of the generated code. The experimental results of the memory usage were encouraging because the examples consumed relatively little RAM. The RAM area is approximately 20% of the ROM area, see Fig.6.9. The memory usage of the device-driver code was presented in total (RAM and ROM), see the column *IO code* of the table. The experimental results showed that the alarm example requires the greatest IO code compared to the others (approx. 11.5 KB). This is because the example makes use of the LCD that requires much more code than other devices such as turning on/off an LED.

General Options	
Target device	NXP LPC2378
Endian mode	little
Output file	executable
C/C++ Compiler Options	
Language	C
C dialect	C99
Language conformance	standard with IAR extensions
Generate interwork code	true
Processor mode	ARM
Optimisation level	high, balanced

Tab. 6.3: IAR C compiler options.

Additionally, the ISR time has been estimated empirically for each case study with expected uncertainty of about $\pm 5 \mu sec$. Although, an analytical study using some static analysis tools (e.g. (Absint, 2012; Tidorum, 2012)) would be interesting to obtain the worst-case execution time of the ISR, such tools are not supported by the current IDE. The ISR times are presented in the last column of Table 6.4. We can notice that the ISR time increases in according to the size of

Case-study		App. Code (Byte)			IO Code (Byte)	ISR
		RAM	ROM	Total	RAM+ROM	time (μs)
Flow1	node_flow (co)	381	2677	3058	2472	25
	node_valve (co)	373	2561	2934	2004	20
Boiler	node_controller (co)	396	2600	2996	1992	40
	node_pump (co)	396	2596	2992	1992	40
	node_waterlevel (co)	408	2836	3244	2472	20
Alarm	node_alarm (rr)	1324	5332	6656	11557	150
ABS	node_control (co)	832	3776	4608	2008	40
	node_wheel0 (co)	506	2729	3235	2480	30
	node_wheel1 (co)	506	2729	3235	2480	30
	node_wheel2 (co)	506	2729	3235	2480	30
	node_wheel3 (co)	506	2729	3235	2480	30

Tab. 6.4: Memory usage of the case studies.

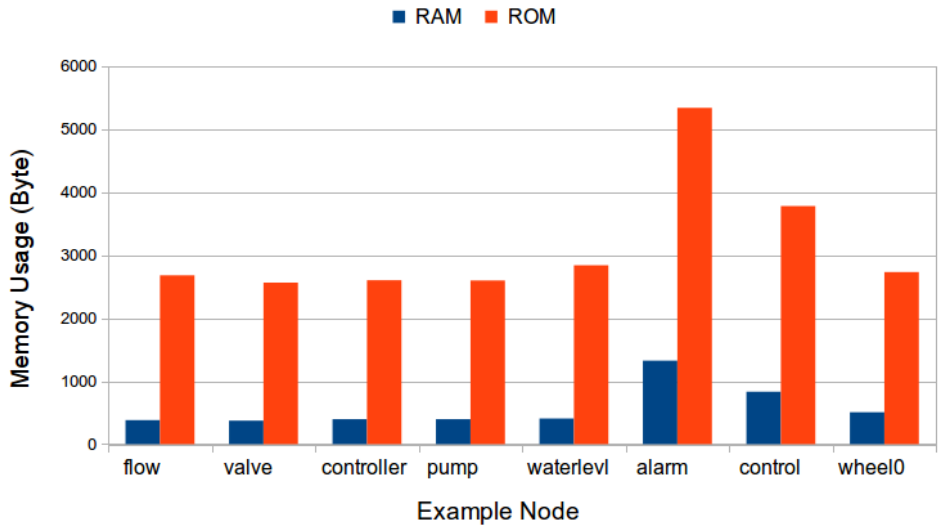


Fig. 6.9: RAM vs. ROM memory usage of the case studies.

the example. The ISR time of the alarm example required considerably longer execution time compared to the other examples because all system components are allocated on the same computing node.

Furthermore, the memory requirement of the generated code for some examples have been compared with alternative method that employs the MicroC/OS-II real-time kernel to implement the same examples. The examples that we use are the original flow regulator system (Flow1) and its varied versions (Flow2 and Flow3) that already discussed in section 6.2.1. The examples implemented

in MicroC/OS-II have been obtained from undergraduate student assignments performed in Northumbria University. Table 6.5 shows the measurements of the memory usage for the two implementation methods. The device driver code was omitted in this experiment since it is fixed in both methods. The results show that our approach can generate C code in which the required memory size is competitive to the traditional method that employs a widely used real-time kernel. Fig. 6.5 presents the results of the comparison. It is apparent that the required RAM of our approach is at least 50% smaller than the required RAM of the MicroC/OS-II implementation.

Case-study		Generated App. Code (Byte)			RTOS App. Code (Byte)		
		RAM	ROM	Total	RAM	ROM	Total
Flow 1	node_flow	381	2677	3058	1000	2402	3402
	node_valve	373	2561	2934	1340	2906	4246
Flow 2	node_flow	485	2821	3306	1164	2472	3636
	node_valve	461	2673	3134	1488	2994	4482
Flow 3	node_flow	589	2953	3542	1368	2538	3906
	node_valve	565	2725	3290	1680	3078	4758
Alarm	node_alarm	1324	5332	6656	2914	5614	8528

Tab. 6.5: Comparison with RTOS code.

6.4 Formal Verification

For the evaluation of the formal verification capability of our approach, we generated TA models from the CANDLE designs of the provided case studies using an existing framework. The main point of this is to discover to what extend the models which are generated are tractable. We verified safety (something wrong never happens) and bounded-response time properties (something useful will happen before some time), see section 6.4.1. Additionally, the model checking tool is used to assess the capability of the available hardware resources. An example of that is a number of transmit/receive buffers required of an employed CAN controller. This demonstrates the usefulness of the generated model during the design phase, see section 6.4.2.

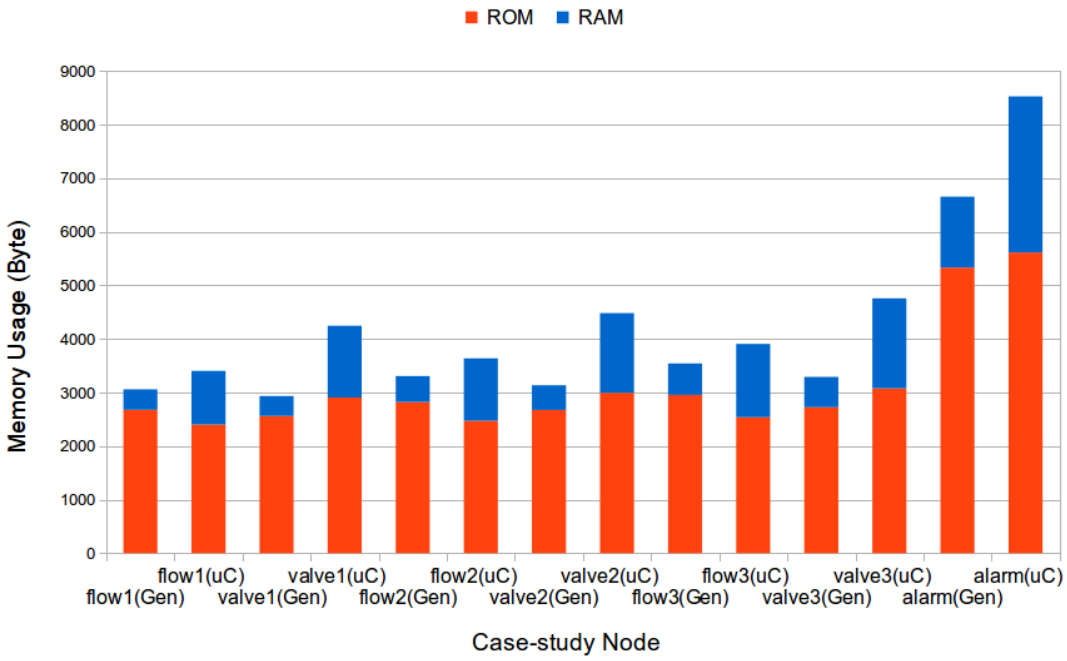


Fig. 6.10: Memory usage comparison.

We used the UPPAAL model checker version 4.0.11 with the verification setting shown in Table 6.6. The model checker program was run on a PC with an AMD Athlon(tm) 3GHz Dual Core Processor 5200B and 2 GB main memory running a Linux operating system.

Option	**Value**
Search Order	Breadth First
State Space Reduction	Conservative
State Space Representation	DBM
Diagnostic Trace	None
Extrapolation	Automatics
Hash table size	16MB

Tab. 6.6: UPPAAL verification options.

6.4.1 Model Checking Properties

Flow Regulator System

For the flow regulator example, we verified the following properties:

P1.1 deadlock freedom,

P1.2 whenever the flow sensor is read, the valve is adjusted within t time units,

P1.3 whenever the flow sensor is read, it is read again within t time units,

P1.4 whenever the valve is adjusted, it is adjusted again within t time units,

P1.5 whenever the flow message is enabled, then it will be received within t time units.

The property P1.1 is a safety property. The property P1.2, P1.3, and P1.4 are bounded-response time properties. They required a separate test automaton to be expressed in the UPPAAL language. The property P1.5 represents the worst-case transmission time of the message. It is also a bounded-response time property. The specification and the verification of the properties are illustrated in the following.

The property P1.1 can be expressed easily in the UPPAAL query language as follows:

```
A[] not deadlock                                                    (P1.1)
```

The property P1.2 is an example of bounded-response time property. The property has the following general form:

$$\forall\Box(P \rightarrow \forall\Diamond Q \wedge g \leq T)$$

which means that whenever P (*request*) is satisfied at a certain time, then Q (*response*) will eventually be satisfied within T time units, where g is a global

clock which is reset once P has occurred. To express the property we, following the approach of (Jensen et al., 1996; Aceto et al., 1998), introduce a separate test automaton that probes the system processes. In the system model, the edges that have observable computations are provided with probe actions. The test automaton is designed to enter a new location when an observable computation is fired. Then a simple form of liveness property (something useful will happen) in UPPAAL is used to construct the property.

The test automaton of the property P1.2 is depicted in Fig 6.11. The test automaton interacts with the system model by using two synchronisation actions: $ReadSensor?$ and $AdjustValve?$. The corresponding actions $ReadSensor!$ and $AdjustValve!$ are added into the edges that contain the computation $ReadSensor$ and $AdjustValve$ respectively. The TA model of the *Flow* and *Valve* process are presented in Appendix B. For example, when the system fires the observable computation $ReadSensor$, the test automaton is forced to enter the location $L1$.

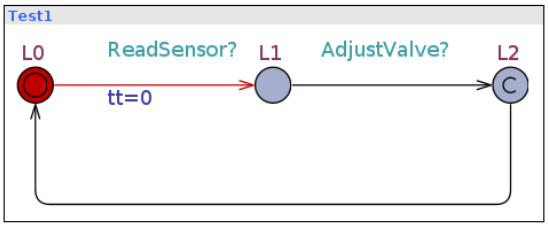


Fig. 6.11: Test automaton of the property P1.2 of the flow regulator example.

The properties are translated into a liveness property in UPPAAL, written $\varphi \rightsquigarrow \psi$, which is read as whenever φ is satisfied, then eventually ψ will be satisfied. φ event means that the test automaton reaches the state $L1$ and ψ event means that the automaton reaches the state $L2$. In order to measure the time required between the two events, we use a clock variable tt. The clock variable is reset whenever the system enters $L1$ state. Moreover, the location $L2$ is chosen to be *committed* which means that time is a not allowed to progress in this location, and the outgoing edge of the location must be involved in the

next state transition. This is very useful when construct the property in order to capture the moment at when the test automaton first enters this location. Then we can write property P1.2 in UPPAAL as follows:

```
(Test1.L1 and Test1.tt == 0) -->
                        (Test1.L2 and Test1.tt <= C1)      (P1.2)
```

where C1 is the upper bound time of the property and it is obtained by trial and error. We have exploited the UPPAAL simulator to obtain the value of C1. First, we run the verifier on a property that simply demonstrates the location $L2$ is reachable in order to get the initial value of C1:

```
E<>(Test1.L2)
```

When the verifier returns, we read the maximum value of the clock variable `Test1.tt` from its range in the simulator window. We then assign this value to C1 and run the property P1.2. If the property is satisfied, we can conclude the final value of C1. If the property is not satisfied, we enable the diagnostic trace in the simulator window and read the maximum value of `Test1.tt` from its range that violates the property. We use the new value to run the property P1.2 again. We repeat this process until the property P1.2 becomes satisfied for a particular value of C1.

The properties P1.3 and P1.4 represent the periodicity of executing the operation *ReadSensor* and $AdjustValve$ respectively. It is useful to predict the worst case time between two consecutive activations of an operation in the system. For example, the operation *ReadSensor* can sense a change in the value of the flow rate only at regular intervals (i.e. the flow rate is sampled). The change in the value may happen just after the completion of the operation. Therefore, to calculate the maximum time to adjust the valve for a new value of the flow rate, we have compute the maximum interval time of the operation *ReadSensor*

and the maximum time between the *ReadSensor* and *AdjustValve* operation. This requirement can be expressed simply by combining the property P1.3 and P1.2.

We use the same approach above to construct properties P1.3 and P1.4. However, the second observable action in the test automaton is replaced with the action that is enabled when the operation is executed again. For example, the test automaton in Fig. 6.12 observes the two consecutive occurrences of the operation *AdjustValve*, so the probe action *AdjustValve*? is used in the first two edges of the automaton. The property is written in UPPAAL as follows:

```
(Test2.L1 and Test2.tt == 0) -->
                        (Test2.L2 and Test2.tt <= C2)      (P1.3)
```

where C2 is the upper bound time of the property and it is obtained by trial and error as well.

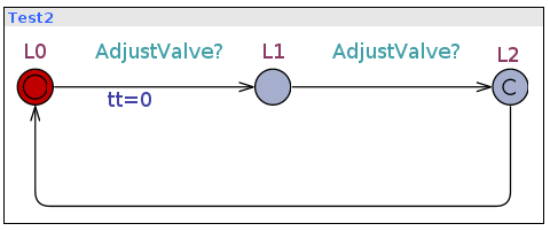


Fig. 6.12: Test automaton of the property P1.3 of the flow regulator example.

The test automaton of the property P1.4 is shown in Fig 6.13 and similarly we can write the property as follows:

```
(Test3.L1 and Test3.tt == 0) -->
                        (Test3.L2 and Test3.tt <= C3)      (P1.4)
```

where C3 is the upper bound time of the property obtained also by trial and error.

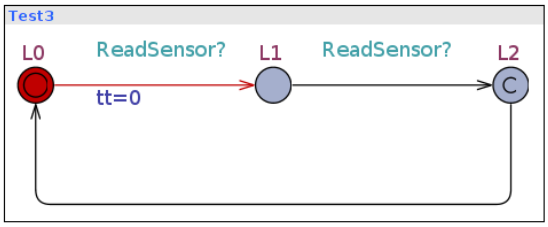


Fig. 6.13: Test automaton of the property P1.4 of the flow regulator example.

The property P1.5 represents the worst-case response time (WCRT) of a message delivered by the CAN. This time includes the queueing time in a transmit buffer and the transmission time of the message. Comparing to traditional methods (e.g, (Davis et al., 2007)), our approach does not require a complex analysis and restricted assumptions (e.g, periodic messages) in order to calculate the WCRT of a message. The approach benefits of the model-checking tool to predict the WCRT of (periodic or aperiodic) messages as long as a TA model of the system is available.

We use the same approach to construct the property. The test automaton observes the communication model of the system. The provided framework generates this model in parallel with the model of the system processes from the design. Appendix B shows the TA model of the communication. The test automaton of the property is depicted in Fig. 6.14. In this figure, the action $k\_e?$ is enabled whenever a message is queued for transmission. The action $k\_a?$ is enabled when the message is accepted. The variable k_v denotes the message identifier (e.g, $FLOW$). A guard (e.g, $k\_v == FLOW$) is added into the first two edges of the automaton to ensure that the transition is enabled only for a particular message id. The property can be written:

```
(Test4.L1 and Test4.tt == 0) -->
                        (Test4.L2 and Test4.tt <= C4)      (P1.5)
```

where C4 is the upper bound time of the property obtained by trial and error.

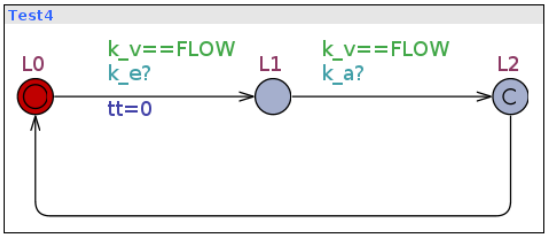


Fig. 6.14: Test automaton of the property P1.5 of the flow regulator example.

The UPPAAL model checker was run in command-line mode because the GUI version does not provide the verification time and memory usage needed to obtain the verification results. If flow_model.xml contains the TA model generated for the flow example, and flow_model.q is a file containing a statement of a property, the property can be checked in UPPAAL using the command:

```
verifyta -u flow_model.xml flow_model.q
```

The verification time and the memory usage needed to check the property are computed using the Memtime tool (Bengtsson, 2012) with version 1.3 which is the default performance measuring tool used by the UPPAAL development team:

```
memtime verifyta flow_model.xml flow_model.q
```

The result when model checking the properties are presented in Table. 6.7. In this table, the second column shows the time needed to check the property in seconds. The third column shows the number of states generated to verify the property. The fourth column shows the size of memory used to verify the property. The result of the verification is shown in the fifth column. The symbol $\surd$ is used to represent that the property is satisfied, the symbol $\times$ is used when the property is unsatisfied, and the symbol ? is used for unknown result. The last column shows the upper bound of the time `Ci` required for a property. The values of `Ci` have been obtained for the ISR period $T = 1000\mu s$ and the ISR time $Cs = 250\mu s$ with CAN bus equals 100 kbit/s. These value

are also assumed for the remaining case studies. Notice that the first property (deadlock freedom) has no time bound, so we show the symbol − instead.

The measurements show that the properties required less than $2MB$ of memory to be verified within a fraction of second. However, notice that the values of the CPU time ($0.10secs$) and memory usage ($1988KB$) shown in the table are minimum values that are generated by the Memtime tool.

Property	Time (s)	State	Memory (KB)	Satisfied	Ci (μs)
P1.1	0.10	28	1988	$\surd$	−
P1.2	0.10	28	1988	$\surd$	C1 = 1870
P1.3	0.10	33	1988	$\surd$	C2 = 11600
P1.4	0.10	28	1988	$\surd$	C3 = 11250
P1.5	0.10	28	1988	$\surd$	C4 = 620

Tab. 6.7: Model-checking results of the flow regulator example.

Steam Boiler Control System

For the steam boiler example, we verified the following properties:

P2.1 deadlock freedom,

P2.2 whenever the sensor reads a low level of water, the pump is turned on within t time units,

P2.3 whenever the sensor reads a high level of water, the pump is turned off within t time units.

The property P2.1 is a safety property, whereas P2.2 and P2.3 are bounded-response time properties. The properties are specified and verified similarly to the properties of the flow example. The result of model checking of the properties are presented in Table. 6.8. The measurements show that the properties required less than $2.MB$ of memory to be verified within a fraction of second.

Property	Time (s)	State	Memory (KB)	Satisfied	Ci (μs)
P2.1	0.10	233	1988	$\surd$	–
P2.2	0.10	233	1988	$\surd$	C1 = 2320
P2.3	0.10	233	1988	$\surd$	C2 = 2320

Tab. 6.8: Model-checking results of the steam boiler example.

Security Alarm System

For the alarm example, we verified the following properties:

P3.1 deadlock freedom,

P3.2 whenever a motion is detected, the alarm timer is enabled within t time units,

P3.3 whenever the alarm timer is expired and the correct code is entered, the sounder (LED) alerts (toggle) within t time units,

P3.4 if the sounder (LED) is in alert (toggle) state and the correct code is entered, the sounder (LED) is turned off within t time units.

The property P3.1 is a safety property, whereas P3.2, P3.3, and P3.4 are bounded-response time properties. The properties are specified and verified similarly to the properties of the flow example. The result of model checking of the properties are presented in Table. 6.9. The measurements show that the properties require less than $70.MB$ of memory to be verified in less than $15sec$.

Property	Time (s)	State	Memory (KB)	Satisfied	Ci (μs)
P3.1	13.42	183942	47160	$\surd$	–
P3.2	12.62	188328	64632	$\surd$	C1 = 1350
P3.3	11.42	184056	64240	$\surd$	C2 = 2920
P3.4	12.21	183944	64372	$\surd$	C3 = 2920

Tab. 6.9: Model-checking results of the security alarm example.

Anti-lock Braking System

For the ABS example, we verified the following properties:

P4.1 deadlock freedom,

P4.2 if the acceleration is changed, the brake pressure of the wheels are regulated within t time units,

P4.3 whenever the first request of speed is sent to a wheel, the brake pressure of the wheels are regulated within t time units.

The property P4.1 is a safety property, whereas P4.2 and P4.3 are bounded-response time properties. The properties are specified and verified similarly to the properties of the flow example. The result of model checking of the properties are presented in Table. 6.10. The measurements show that the properties required less than $60.MB$ of memory to be verified within a few seconds. This demonstrates a potential advantage of our approach since complex systems can be expressed by quite small models. By contrast, in the approach of (Herber, 2010), the UPPAAL model of the ABS system generated from SystemC code has approximately 10 times more *processes* than the UPPAAL model generated by our approach from the CANDLE program which implements the same functionality. Table 6.11 summarises the main characteristics of the two TA models of the ABS example. This has a considerable impact on the tractability of the generated model. For instance, the UPPAAL model checker has been used to check a basic property (deadlock freedom) on the TA model generated from the SystemC code for the ABS example. The verification ended after more than 5 hours with an *out-of-memory* error and 16777216 states stored during the verification. The same experiment, undertaken on our TA model of the same case study, terminated positively within 0.60 second, having stored 11469 (see Table 6.10).

Property	Time (s)	State	Memory (KB)	Satisfied	Ci (μs)
P4.1	0.60	11469	38492	$\surd$	–
P4.2	0.60	11469	55408	$\surd$	C1 = 3320
P4.3	1.20	11469	56192	$\surd$	C2 = 14220

Tab. 6.10: Model-checking results of the ABS example.

	ABS TA Model (CANDLE)	ABS TA Model (SystemC)
Process	12	121
Channel	8	168
Clock	0 global and 12 local	1 global and 1 local
Variable	8	283

Tab. 6.11: TA model comparison of the ABS system.

6.4.2 Verifying Transmit/Receive Buffer Resources

It is very useful for system developers to be able to assess the capability of the available hardware that will be used to run the system. An example is the number of transmit/receive buffers of the CAN controller required by the system implementation. For that reason, TA models have been generated for varied versions of the flow example because it was simple to change the complexity of the example. The example was varied by adding new processes of *Flow* and *Valve* to the original design in according to the second scenario discussed in section 6.2.1. In this scenario the system architecture consists of two computing nodes sharing a single CAN bus. The Flow processes are allocated to the first node, and the Valve processes are allocated to the second one. The employed CAN controller (NXP, 2009) features triple transmit buffers and double receive buffers. The UPPAAL model checker was used to determine the number of transmit/receive buffers required by the examples in the worst-case.

Firstly, to determine the maximal number of the transmit buffers, the TA model of the CAN shown in Appendix B is amended as follows. Whenever a new message is enqueued, a counter is incremented. Each node transmits a particular set of messages which can be known at design time. Therefore, if we wish to calculate the number of the transmit buffers for a particular node, we have to

constrain the operation of the counter increment. For example:

```
if((k_v==id1)||(k_v==id2)||(k_v==id3)){
  tCounter++;
}
```

where k_v holds the current message ID (id1, id2, or id3) that can be sent by the node. tCounter is a variable that holds the number of the transmit buffers. When the message is successfully transmitted, the counter is decremented. This happens when the model fires the post-acceptance transition. Then the property is expressed in UPPAAL as:

```
A[] (k.tCounter <= C1)
```

The property will be satisfied if the number of buffers never exceeds the value C1 which denotes the number of the available transmit buffers.

Secondly, to determine the number of the receive buffers, we introduced a separate TA model, see Fig.6.15. The model consists of three locations, $L0$, $L1$ and $L2$. It is possible to move from location $L0$ to $L1$ at any time. Once the automaton enters $L1$, it can stays in this location up to $(T + Cs)$ time units. This time expresses the maximum interval time that a message may wait in the buffer until it is polled by the ISR. This time equals the ISR period (T) plus the ISR execution time (Cs). A counter is incremented whenever a new message from a predefined set is accepted. A particular node can receive a predefined set of messages. The counter is used to hold the number of buffers. The UPPAAL property:

```
A[] (rBuff.rCounter <= C2)
```

is true if the number of buffers never exceeds the value C2, where C2 expresses the number of the available receive buffers.

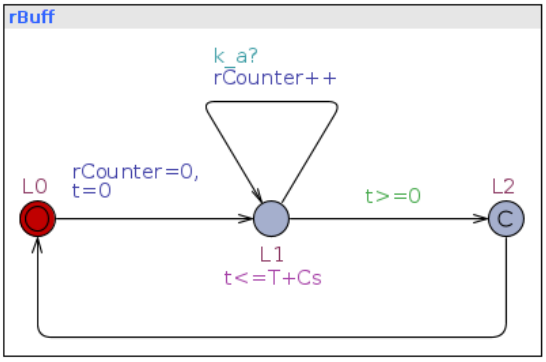


Fig. 6.15: Test automaton of receive buffer verification.

The results of the transmit/receive buffer verification are presented in Table 6.12 and Table 6.13 respectively. The verification was run for the following setting: $T = 1000\mu s$, $Cs = 250\mu s$, the transmission rate of $200kbit/s$, and a message payload of $1byte$. In these tables, the first column shows the number of *Flow* and *Valve* processes used in the example, the second column shows the available number of buffers. The computation resources (CPU time, state-space size, and memory usage) to run the verification are shown in the next three columns respectively. Finally, the result of the verification is shown in the last column.

The experiments showed that it is possible to verify the transmit buffer number of 15 processes of Flow and Valve in the current computing resources. It took less than 4 minutes of CPU time, and approximately $110MB$ of memory usage, see Table 6.12. However, the number of receive buffer was verified for the examples with only up to 12 processes. The example with 12 processes took about 75 minutes of CPU time, and $800MB$ of memory usage, see Table 6.13. The verification of 13 processes was terminated after 16 hours of running the experiment with reported memory usage exceeds $2GB$. The experiment took an unexpectedly long time because the model checker program made use of the swap memory. When a state-space size becomes larger than an available memory, a part of the state space is usually stored in a swap area. It is well known that reading/writing operations from/to the swap memory are slower than that

performed in the main memory. The observed time and space complexity of the verification of receive buffer is shown in Fig. 6.16 and Fig. 6.17 respectively. The figures show that the CPU time and memory usage are increased rapidly as the number of processes is increased.

Process	C1	Time (s)	State	Memory (KB)	Satisfied
1	3	0.10	43	2008	√
2	3	0.10	284	2008	√
3	3	0.10	1030	2008	√
4	3	0.10	457	2008	×
5	3	0.20	1266	37840	×
6	3	0.30	2999	37984	×
7	3	0.70	6367	38108	×
8	3	1.60	12444	38508	×
9	3	3.90	22786	39296	×
10	3	8.41	39573	40376	×
11	3	18.02	65776	41988	×
12	3	36.13	105351	44952	×
13	3	69.46	163462	65812	×
14	3	128.82	246735	87364	×
15	3	236.16	363545	112708	×

Tab. 6.12: Transmit buffer verification of the flow regulator examples.

Process	C2	Time (s)	State	Memory (KB)	Satisfied
1	2	0.10	43	2008	√
2	2	0.10	284	2008	√
3	2	0.10	315	2008	×
4	2	0.20	883	37840	×
5	2	0.30	2599	37844	×
6	2	0.80	7887	38112	×
7	2	3.10	24383	39076	×
8	2	13.62	76283	41816	×
9	2	58.76	240519	50632	×
10	2	252.15	762055	112500	×
11	2	1073.58	2420911	289392	×
12	2	4423.76	7698483	814088	×
13	2	≥57615.16	?	2220692	?

Tab. 6.13: Receive buffer verification of flow regulator examples.

The results show that only the examples that have up to three Flow and Valve processes satisfy the available transmit buffer of the employed CAN controller,

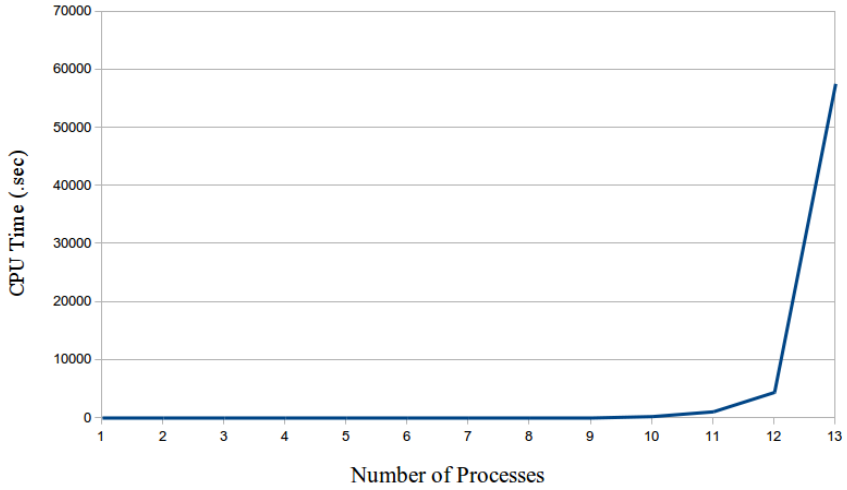


Fig. 6.16: Time complexity of receive buffer verification.

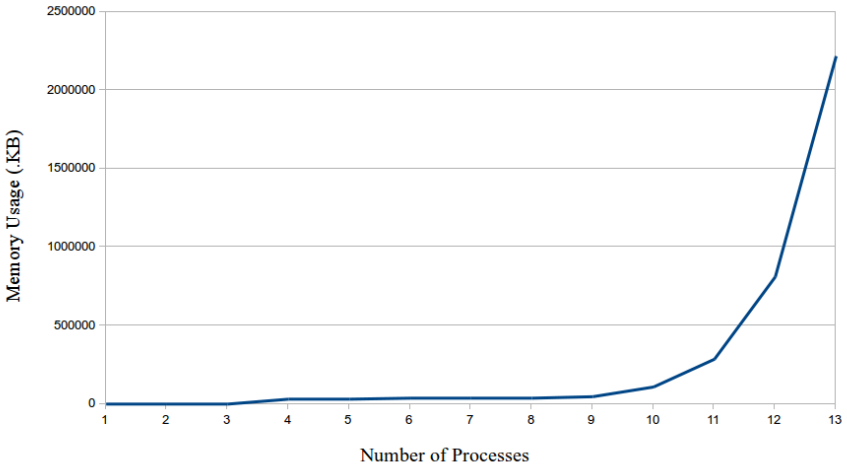


Fig. 6.17: Space complexity of receive buffer verification.

see Table 6.12, whereas the examples with up to two Flow and Valve processes satisfy the available receive buffer, see Table 6.13. This is because the Flow processes have similar behaviour, i.e. they run periodically at the same rate, so they all come to the same instant when they enqueue their messages into the transmit buffer. A way of investigating this further was considered by adding an offset to each Flow process, so they run at different rates. The offsets were assigned as follows. If there are N Flow processes in the example, the first Flow process will have 1 tick offset, the second will have 2 ticks offset, ... and so on. Interestingly, this modification enables the examples to satisfy both the available transmit and receive buffer for some examples, see Table 6.14 and Table 6.15 respectively. In the first experiment, we verified the number of transmit buffers for 11 processes of Flow and Valve and the verification returns positive results for this examples. The 12 processes example however was not determined. The verification for this example was terminated after approximately 65 hours, see Table 6.14. In the second experiments, we were able to verify the number of receive buffers for 14 processes in the current computing resources. For instance, the verification of 14 process took less than 6 hours to return the result, see Table 6.15. However, the verification of 15 processes was terminated after about 16 hours and so the result was not determined.

Process	C1	Time (s)	State	Memory (KB)	Satisfied
1	3	0.10	48	2008	√
2	3	0.10	278	2008	√
3	3	0.20	906	37832	√
4	3	0.20	2669	37844	√
5	3	0.60	6765	38112	√
6	3	1.70	16041	38644	√
7	3	5.01	37662	39692	√
8	3	15.12	87949	42232	√
9	3	45.34	203551	47924	√
10	3	148.86	502548	62684	√
11	3	1680.50	3316922	220852	√
12	3	≥235006.00	–	–	?

Tab. 6.14: Transmit buffer verification of flow regulator examples with offset.

Process	C2	Time (s)	State	Memory (KB)	Satisfied
1	2	0.10	48	2008	√
2	2	0.10	278	2008	√
3	2	0.10	906	2008	√
4	2	0.20	2669	37848	√
5	2	0.60	6765	38116	√
6	2	1.60	16041	38508	√
7	2	5.01	37662	39692	√
8	2	15.13	87949	42232	√
9	2	45.35	203551	47932	√
10	2	148.79	502548	62680	√
11	2	560.68	1137807	97684	×
12	2	4247.32	4844946	309928	×
13	2	8213.91	10231159	623748	×
14	2	19925.65	21003639	1233312	×
15	2	≥57996.51	–	2081260	?

Tab. 6.15: Receive buffer verification of flow regulator examples with offset.

Surprisingly, only the verification of the examples with up to 10 processes returns positive results. The examples with more than 10 processes do not satisfy the number of the available receive buffer. The counter-example generated by the model checker program was analysed to figure out the source of the problem. The investigation leads to a problem in our policy of assigning offsets. Although each Flow process runs periodically with different offset, the offset value should not exceed the period of the process. The current period of Flow processes is 10 ticks. So the process 11 will coincide with the process 1, the process 12 will coincide with the process 2, ... and so on. This explains the unsatisfied result of the verification for the examples with more than 10 processes.

Alternatively, we could modify the system architecture by employing more computing nodes. Then we can allocate a fewer number of Flow and Valve processes in each node. This will decrease the number of transmit/receive buffer required by each node. Although this solution has an impact on the cost of the system implementation, it could be more feasible in the cost than using higher performance hardware to comply with the provided design. Consequently, the experiments demonstrate the usefulness of having a model of the system early

during the design phase, because the user can then verify the applicability of the available resources before running the system implementation.

6.5 Summary

The experiments have successfully demonstrated the feasibility, the performance and the formal verification capability, of our code generation approach. The computational time of the transformation from CANDLE to executable C code and UPPAAL model is modest and varies only a little as a function of the number of processes, the code size, and the net size. It requires only a few seconds even for large examples. The efficiency of the generated code has been compared with other code that employs a real-time kernel. The measurements have shown that our generated code is competitive in size to the version implemented by a widely-used RTOS, and the required RAM is at least 50% smaller in size than that is required by the RTOS. This demonstrates that our approach can generate an efficient code suitable for limited resource embedded systems. Additionally, the generated code is guaranteed to realise the behaviour of CANDLE model which can be formally verified against key properties. The experimental results of the formal verification capability are promising. A number of useful properties of a broad class of interesting examples have been successfully verified within acceptable computation resources. The results have shown that using a model-checking tool is very useful during the design time to assess run-time resources requirements.

7. CONCLUSIONS AND FUTURE WORK

In this chapter, we present a summary of the contributions of the thesis, discuss the limitations of the work, and present possible directions for future research.

7.1 Summary of Contributions

The work presented in this thesis addressed the problem of generating executable code for CAN-based distributed embedded systems in a way that guarantees that both functional and timing properties expressed in a high level formal language are satisfied. The thesis proposed a novel approach in which system behaviour is specified in CANDLE, a high-level language which is given a formal semantics by translation to bCANDLE, an asynchronous process calculus. A bCANDLE system is translated automatically, via a common intermediate net representation, both into executable C code and into timed automaton model that can be used in the formal verification of a wide range of functional and temporal properties.

A code generator was developed that can automatically produce executable C code from a CANDLE specification. The code is generated from CANDLE via an intermediate net representation. An efficient C representation of the net was presented. We introduced a time-triggered execution (implementation) model to execute the net in which, at each tick, a scheduler determines which computation should run next. The schedulers that we have considered include

simple round-robin, weighted round-robin, cooperative, and hybrid methods. The hybrid scheduler combines cooperative and round-robin scheduling techniques. These scheduling strategies facilitate off-line prediction of the worst-case response time of system computations. A single broadcast asynchronous communication mechanism was adopted and implemented. The communication mechanism is an abstraction of the CAN. All communications occur through this mechanism and never through the use of shared variables. This single notion, employed both for external and local communication between system components, provides flexibility to the system developer to freely distribute system components on a number of nodes, and simplifies the process of generating a formal model of the system. An AADL-like language was introduced to describe the system architecture. The description file provides details to the code generator about processes, nodes, process-to-node allocation, scheduling algorithm, tick rate, and communication details, including the IDs of messages and network transmission rate.

The bCANDLE semantics assumes that computations complete and update their data instantaneously and atomically on completion. Therefore, a number of methods were proposed that ensure the atomicity of data update. The methods were evaluated based on three criteria we identified. It was concluded that for short computations that complete within one tick, the method *on-tick duration computation* would be suitable, and for long running computations, the *delayed atomic update* method would be considered for our code generator.

A rigorous argument was presented that, for any system expressed in the high-level language, its formal model is a conservative approximation of the executable C code. This allows the system developer to conclude that if a model satisfies any universally quantified property, then it is guaranteed that the implementation will also satisfy the same property.

A variety of experiments were conducted to assess the applicability and per-

formance of our approach. We used four representative case studies: a flow regulator control system, a boiler control system, a security alarm system, and an anti-lock braking system. Executable C code and formal models were successfully generated for the case-studies in a reasonable time. The memory consumption of the generated code was assessed and compared with an alternative method employing a widely-used real-time kernel. The experimental results showed that overall memory consumption of generated code is competitive with that used by the real-time kernel. Additionally, the results showed that RAM usage is at least 50% less than that required by the real-time kernel. These results give us confidence in the viability of implementing a code generator based on the net representation of the CANDLE system under development. The tractability of the model checking problem for the generated formal models was assessed by using an off-the-shelf model checker. A number of useful properties were successfully verified within acceptable computation resources. The experimental results showed that the generated models are comparatively small. We believe this demonstrates the power of the abstractions adopted by our approach. The results showed also that using a model-checking tool is very useful during system design phase to assess run-time resource requirements in addition to typical functional and temporal properties.

This is the first time that a code generator for CAN-based systems has been developed that allows model checking of specifications that can be guaranteed to retain the behaviour of implementations.

7.2 Limitations

The main motivation behind our approach is the capability of generating a model amenable to model checking and executable C code suitable for implementation. An informal argument was proposed to confirm that the formal model conservatively approximates its implementation. This work did not for-

mally verify the relationship between the model and the code. The reason for that is that the target implementation language (which is C) lacks a formal semantics. Moreover, further validation would be required also for the translation from C to a low-level machine language. This is important to ensure that the behaviour described in C is preserved during the compilation to the target machine code. A rigorous solution to this problem is not trivial. The problem has been addressed in the *CompCert* project (CompCert, 2012) and the results have been extensively published, for instance (Blazy, 2008; Dargaye, 2009; Leroy, 2009; Blazy and Leroy, 2009; Bedin et al., 2012).

Additionally, a model checking technique was used to verify that the model exhibits some useful properties. The main drawback with such a technique is the state-space explosion. This work did not investigate methods and techniques to tackle this problem. However, computer capacity has been significantly improved in terms of processing speeds and main memory sizes. This allows user to tackle larger problem sizes that were previously impossible to verify. Furthermore, model checking algorithms and tools are also exploiting the trend in increasing hardware performance. An example of this is the capability of executing a model checker on multi-core machines or a network of computers. For instance, the Spin model checker has been recently enhanced to support such features, see (Holzmann et al., 2011; Holzmann, 2012).

7.3 Future Work

Preliminary work has been conducted. The experimental results demonstrate the viability of the proposed approach. Its performance is adequate and promising. In the following, a number of avenues of future investigation are suggested.

A number of different case studies were successfully applied in this work to demonstrate the practical applicability of our approach. An interesting avenue

of further investigation is to push our approach to the limit by applying larger case studies. This would be very useful to explore the industrial strength of the approach when it is applied to industrial examples. Then we could gain a better idea about the capability of the approach, including code and model generation.

A time-triggered software architecture was employed to execute the net in which a single periodic interrupt is allowed in the system implementation. Currently the interrupt service routine (ISR) is modelled implicitly avoiding a detailed modelling of the ISR. This simplifies our models and makes model checking more tractable. This however introduces pessimism into the models by introducing wider time bounds on system computations. One direction for future work would be investigating the impact of including the ISR details explicitly in the models. This can yield narrower bounds but the verification can become more expensive since the size of the state space is expected to increase. Modelling the ISR explicitly can also provide more flexibility in the selection of scheduling algorithms because this allows modelling the scheduler in more detail and thereby makes a wider range of schedulers available to the code generator.

Another avenue of investigation involves the application of software model checking directly on the generated code. This would require applying abstraction methods on the code such as counterexample-guided abstraction refinement (CEGAR) (Clarke et al., 2000). Then it would be interesting to compare the tractability of our generated code to an existing software model checking tool with that of code that is hand-crafted and implements the same functionality. This would evaluate the generated code amenability for software model checking technique.

A number of methods were proposed that ensure the atomicity of the data update. The current case studies applied in the experimental chapter are not affected by the problem of data interruption because computations are either

short that complete within one tick, or long running that are not interrupted by other computations. Future work on this work would require implementing a chosen solution for the code generation, and testing it in different case studies that could encounter this problem.

The intermediate net representation of CANDLE is the base of the model generator and the code generator. One possible direction of further research is to investigate optimisation techniques that can be applied to reduce the size of the net. For example, in the translation of LOTOS to C programs, a collection of optimisations have been applied on the control and data flow of an intermediate Petri net stage (Garavel et al., 2011). It is interesting to investigate the applicability of such techniques (e.g, (Garavel and Serwe, 2006)) to our net that is generated from bCANDLE. This would contribute both during model-checking and code generation stage. In other words, the optimised net could yield a smaller executable code, and a possible reduction in the size of the formal model. Although some techniques in (Kendall, 2001b) were applied to tackle the problem of the state-space size, this approach could consider the state-space explosion problem early before the model is generated. Additionally, a finite state-machine (FSM) representation can be generated from the net representation in order to generate the executable code. Although we think this would have a considerable impact on the size of the generated code since FSM has a higher representation cost compared to the Petri net, such code would be faster to execute. An example of comparison between FSM and Petri net representation can be found in (Zhu and Brooks, 2009). The system developer may then have an option to select between two alternative methods to generate the code.

For some scenarios, verifying the model of the whole system can be not possible because of the state-space explosion problem. An interesting avenue of future work would be incorporating compositional techniques to handle this problem.

In this approach of analysis, several components of the system can be modelled in more abstract way, and only one component is modelled in detail. After that, we move to the next component and we do the same. If each of these components considered in the abstract environment satisfies some properties, then we can claim that the composition of these components satisfies also the same properties. Some compositional methods have been already applied to bCANDLE in the work of (Brockway, 2010). Therefore, extending the current work to consider such techniques can be straightforward.

We proposed an approach that can automatically generate executable C from specification of CAN-based system. In some industries such as automotive, aerospace, medical devices, and others, a particular software development standard is followed, specifically for those systems that are programmed in C. One such example is the MISRA C standard (MISRA, 2004) which is mainly developed to ensure the products are suitable for application in the automotive industry. The MISRA C standard defines a number of rules (constraints) in using the C language on safety-related systems. We think that it is interesting for the industry that the code generator are able to produce a C code that conforms to such standard. This feature would improve the code quality and could make the code generator more accepted by the previously mentioned industries.

Multi-core platforms provide high performance computing capability compared to traditional single-core platforms. They integrate many processors on a single chip which are connected through a Network on Chip (NoC). For instance, the Intel Single-Chip Cloud Computer (SCC) comprises of 48 cores on a single chip (Petrović et al., 2012). Our approach facilitates executing a number of processes in a single-processor platform via using the concept of local channel introduced in Chapter 3. The local channel has been already implemented using shared memory where the processes communicate by message passing. We think it would be straightforward to extend this concept to leverage the power

of multi-core platforms. Then each process could be executed on a separate processor (core) and communicate via local channels. This can offer an alternative implementation option to the system developer to improve the overall performance of the system implementation.

The CAN is the dominant network in automotive and factory control systems and is becoming increasingly popular in robotic, medical and avionics applications. There are a wide variety of other broadcast protocols available in practice, each is dedicated for a particular application area. For example, Profibus (Tovar and Vasques, 1999) for process control, LON (Rabbie, 2005) for building automation, and ZigBee (Baronti et al., 2007) for wireless sensor network. It would be interesting to extend the current work to consider more communication protocols than the CAN. The formal language bCANDLE has been constructed with the CAN protocol in mind. The network model of the language defines a number of semantics rules to describe the network behaviour. We think that these semantics rules should be revisited in order to adopt the behaviour of other network protocols.

APPENDIX

A. CASE STUDIES

In this section, we show the CANDLE program, the architecture description, the computations bounds, the bCANDLE model, and the net representation respectively of the examples introduced in Chapter 6.

A.1 Flow Regulator System

CANDLE Program

```
Flow | Valve

where

Flow =
  every 10000 do
    readSensor();
    snd(k, FLOW, fFlow)
  end every

Valve =
  loop do
    rcv(k, FLOW, vFlow);
    adjustValve()
  end loop
```

Architecture Description

```
node flow
  wordsize   : 32
  tickHz     : 1000
  processes  : Flow
  scheduler  : COOPERATIVE
  ports      : CAN_0
```

```
process Flow
  stacksize : 20
  channels  : k -> CAN_0

node valve
  wordsize   : 32
  tickHz     : 1000
  processes  : Valve
  scheduler  : COOPERATIVE
  ports      : CAN_0

process Valve
  stacksize : 20
  channels  : k -> CAN_0

channel k
  bps : 100000
  messages : <FLOW:1>
```

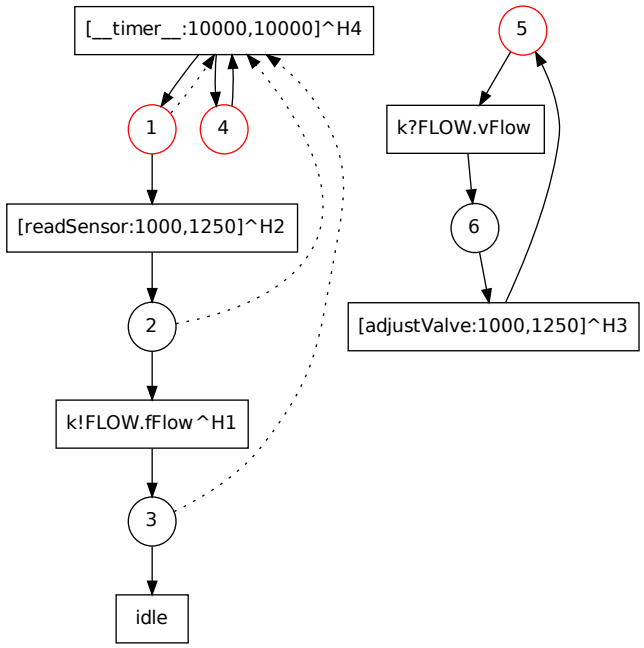
Computations Bounds

```
readSensor              1000  1250
adjustValve             1000  1250
```

bCANDLE Model

```
(Flow | Valve)

where

Flow  = __LOOP__0

Valve = __LOOP__1

__LOOP__0 = (((([readSensor:1000,1250] ; k!FLOW.fFlow) ; idle)
                [> [__timer__:10000,10000]) ; __LOOP__0)

__LOOP__1 = ((k?FLOW.vFlow ; [adjustValve:1000,1250]) ; __LOOP__1)
```

Net Representation



A.2 Steam Boiler Control System

CANDLE Program

```
WaterLevel | Pump | Controller

where

WaterLevel =
  initSensor();
  select
    when rcv(k, START) => null
  in
    every 5000 do
      snd(k, SENSOR_READY)
    end every
  end select;
  select
    when rcv(k, SHUTDOWN) => idle
  in
    every 10000 do
```

```
      case readSensor()
        when 0 => snd(k, LEVEL_OK)
        when 1 => snd(k, LEVEL_LOW)
        when 2 => snd(k, LEVEL_HIGH)
      end case
    end every
  end select

Pump =
  initPump();
  select
    when rcv(k, START) => null
  in
    every 5000 do
      snd(k, PUMP_READY)
    end every
  end select;
  select
    when rcv(k, SHUTDOWN) => pumpOff(); idle
  in
    loop do
      select
        when rcv(k, PUMP_ON) => pumpOn()
        when rcv(k, PUMP_OFF) => pumpOff()
      end select
    end loop
  end select

Controller =
  initController();
  select
    when rcv(k, SENSOR_READY) => rcv(k, PUMP_READY)
    when rcv(k, PUMP_READY) => rcv(k, SENSOR_READY)
  end select;
  snd(k, START);
  loop do
    select
      when rcv(k, LEVEL_OK) => null
      when rcv(k, LEVEL_LOW) => snd(k, PUMP_ON)
      when rcv(k, LEVEL_HIGH) => snd(k, PUMP_OFF)
      timeout elapse(15000) => snd(k, SHUTDOWN); idle
    end select
  end loop
```

Architecture Description

```
node waterlevel
```

```
  wordsize   : 32
  tickHz     : 1000
  processes  : WaterLevel
  scheduler  : COOPERATIVE
  ports      : CAN_0

process WaterLevel
  stacksize : 25
  channels  : k -> CAN_0

node pump
  wordsize   : 32
  tickHz     : 1000
  processes  : Pump
  scheduler  : COOPERATIVE
  ports      : CAN_0

process Pump
  stacksize : 25
  channels  : k -> CAN_0

node controller
  wordsize   : 32
  tickHz     : 1000
  processes  : Controller
  scheduler  : COOPERATIVE
  ports      : CAN_0

process Controller
  stacksize : 25
  channels  : k -> CAN_0

channel k
  bps : 100000
  messages : <START:0, SENSOR_READY:0, SHUTDOWN:0, LEVEL_HIGH:0, LEVEL_LOW:0,
                      LEVEL_OK:0, PUMP_READY:0, PUMP_ON:0, PUMP_OFF:0>
```

Computations Bounds

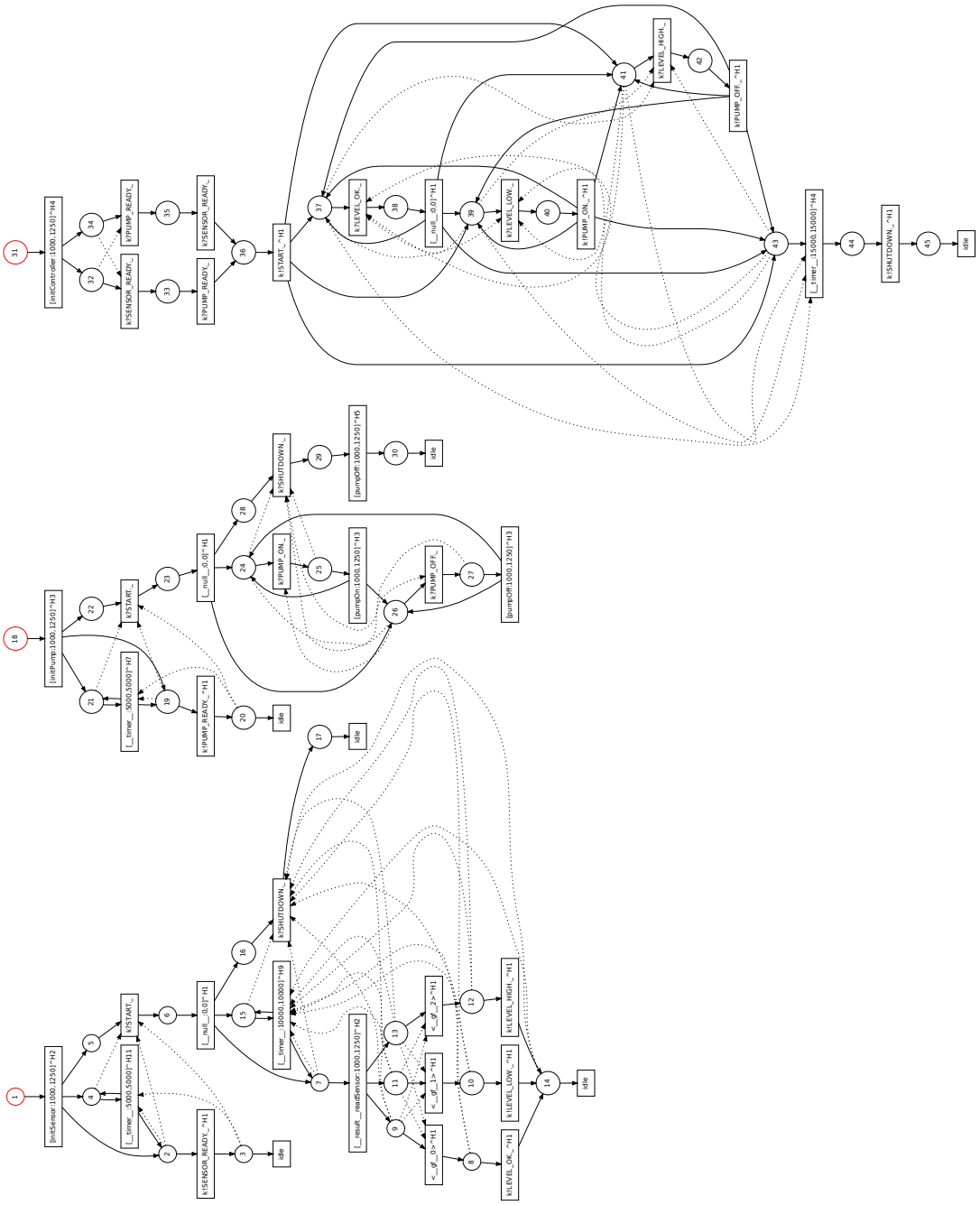
```
initSensor              1000  1250
readSensor              1000  1250
initPump                1000  1250
pumpOn                  1000  1250
pumpOff                 1000  1250
initController          1000  1250
```

```
testLevel               1000  1250
```

bCANDLE Model

```
(WaterLevel | (Pump | Controller))
where
WaterLevel =
  ([initSensor:1000,1250] ;
   ((__LOOP__0 [> (k?START._ ; [__null__:0,0])) ;
    (__LOOP__1 [> (k?SHUTDOWN._ ; idle))))

Pump =
  ([initPump:1000,1250] ;
   ((__LOOP__2 [> (k?START._ ; [__null__:0,0])) ;
    (__LOOP__3 [> (k?SHUTDOWN._ ; ([pumpOff:1000,1250] ; idle)))))

Controller =
  ([initController:1000,1250] ;
   (((k?SENSOR_READY._ ; k?PUMP_READY._) + (k?PUMP_READY._ ; k?SENSOR_READY._)) ;
   (k!START._ ; __LOOP__4)))

__LOOP__0 =
  (((k!SENSOR_READY._ ; idle) [> [__timer__:5000,5000]) ; __LOOP__0)

__LOOP__1 =
  (((([__result__readSensor:1000,1250] ;
   ((__gf__0 -> k!LEVEL_OK._) + ((__gf__1 -> k!LEVEL_LOW._) +
    (__gf__2 -> k!LEVEL_HIGH._)))) ; idle)
   [> [__timer__:10000,10000]) ; __LOOP__1)

__LOOP__2 =
  (((k!PUMP_READY._ ; idle) [> [__timer__:5000,5000]) ; __LOOP__2)

__LOOP__3 =
  (((k?PUMP_ON._ ; [pumpOn:1000,1250]) + (k?PUMP_OFF._ ; [pumpOff:1000,1250])) ; __LOOP__3)

__LOOP__4 =
  (((k?LEVEL_OK._ ; [__null__:0,0]) + ((k?LEVEL_LOW._ ; k!PUMP_ON._) +
   ((k?LEVEL_HIGH._ ; k!PUMP_OFF._) +
   ([__timer__:15000,15000] ; (k!SHUTDOWN._ ; idle))))) ; __LOOP__4)
```

Net Representation


[initSensor:1000,1250]^H2
[initPump:1000,1250]^H3
[initController:1000,1250]^H4
[__timer__:5000,5000]^H1
[__timer__:5000,5000]^H7
[__timer__:10000,10000]^H9
[__timer__:15000,15000]^H4
[__resul__:readSensor:1000,1250]^H2
[pumpOn:1000,1250]^H3
[pumpOff:1000,1250]^H5
[__null__:0,0]^H1
[__null__:0,0]^H1
[__null__:0,0]^H1
<__qf__1>^H1
<__qf__2>^H1
<__qf__3>^H1
k!START_
k!SENSOR_READY_
k!PUMP_READY_
k!SHUTDOWN_
k!LEVEL_OK_
k!LEVEL_LOW_
k!LEVEL_HIGH_
k!PUMP_ON_
k!PUMP_OFF_
k!START_^H1
k!SENSOR_READY_^H1
k!PUMP_READY_^H1
k!SHUTDOWN_^H1
k!LEVEL_OK_^H1
k!LEVEL_LOW_^H1
k!LEVEL_HIGH_^H1
k!PUMP_ON_^H1
k!PUMP_OFF_^H1
idle

A.3 Security Alarm System

CANDLE Program

```
Control | PendingTimer | Flasher | Display
where
Control =
  loop do
    select
      timeout elapse(50000) => snd(k, CONTROL, controlMsg)
    in
      select
        when rcv(k, PENDING_TIME, controlTimeLeft) => controlStatePendingTime()
      in
        case control()
          when 0 => null
          when 1 => snd(k, PENDING, controlTimeInterval)
          when 2 => snd(k, ALERT)
          when 3 => snd(k, ABORT)
        end case;
        idle
      end select;
      idle
    end select
  end loop
PendingTimer =
  loop do
    rcv(k, PENDING, ptTimeLeft);
    trap
      when PENDING_DONE => null
    in
      select
        when rcv(k, ABORT) => exit PENDING_DONE
      in
        every 1000000 do
          case pendingExpired()
            when true =>  snd(k, PENDING_TIME, ptTimeLeft); exit PENDING_DONE
            when false => snd(k, PENDING_TIME, ptTimeLeft)
          end case
        end every
      end select
    end trap
  end loop
Flasher =
  loop do
    rcv(k, ALERT);
    select
```

```
      when rcv(k, ABORT) => linkLEDOff()
    in
      every 250000 do
        linkLEDToggle()
      end every
    end select
  end loop
Display =
  initDisplay();
  loop do
    rcv(k, CONTROL, displayMsg);
    Display()
  end loop
```

Architecture Description

```
node alarm
  wordsize   : 32
  tickHz     : 1000
  processes  : Control, PendingTimer, Flasher, Display
  scheduler  : ROUND_ROBIN
  ports      : LOCAL_DATA

process Control
  stacksize : 25
  channels  : k -> LOCAL_DATA

process PendingTimer
  stacksize : 25
  channels  : k -> LOCAL_DATA

process Flasher
  stacksize : 25
  channels  : k -> LOCAL_DATA

process Display
  stacksize : 128
  channels  : k -> LOCAL_DATA

channel k
  bps : 100000
  messages : <CONTROL:8, PENDING_TIME:8, PENDING:8, ALERT:8, ABORT:8>
```

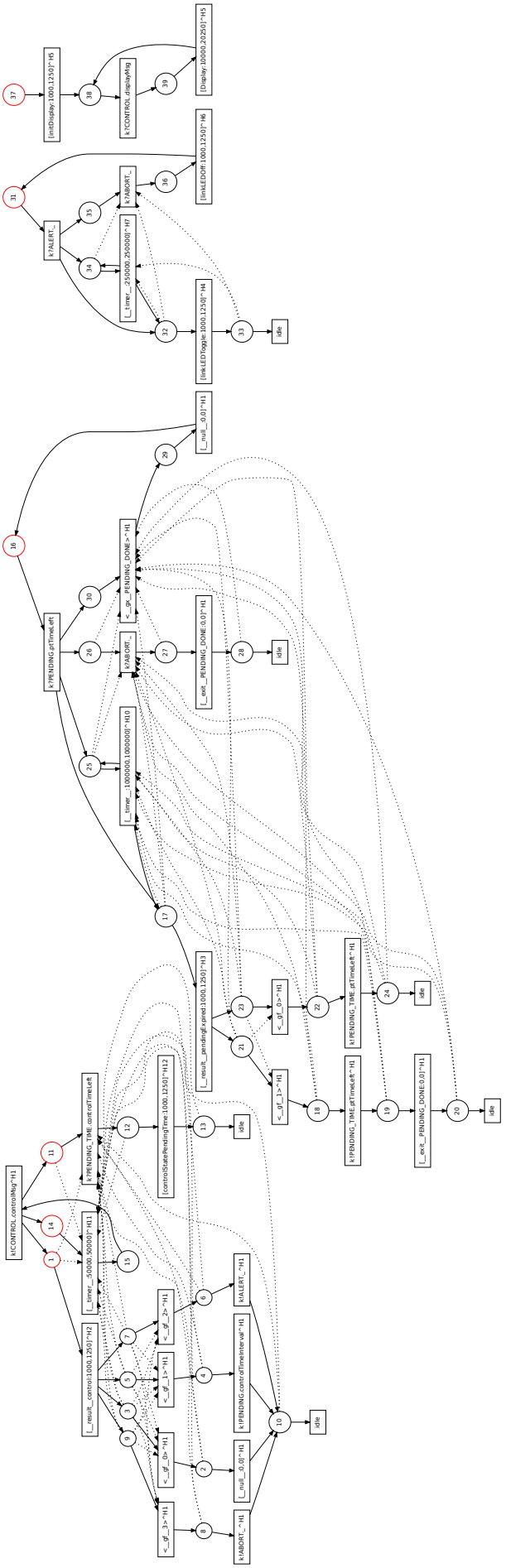
Computations Bounds

```
controlStatePendingTime  1000   1250
control                  1000   1250
pendingExpired           1000   1250
linkLEDOff               1000   1250
linkLEDToggle            1000   1250
initDisplay              1000   1250
Display                 10000  20250
```

bCANDLE Model

```
(Control | (PendingTimer | (Flasher | Display)))

where

Control = __LOOP__0
PendingTimer = __LOOP__1
Flasher = __LOOP__3
Display = ([initDisplay:1000,1250] ; __LOOP__5)

__LOOP__0 =
  ((((((__result__control:1000,1250] ;
     ((__gf__0 -> [__null__:0,0]) + ((__gf__1 -> k!PENDING.controlTimeInterval) +
     ((__gf__2 -> k!ALERT._) + (__gf__3 -> k!ABORT._))))) ; idle)
    [> (k?PENDING_TIME.controlTimeLeft ; [controlStatePendingTime:1000,1250])) ; idle)
   [> ([__timer__:50000,50000] ; k!CONTROL.controlMsg)) ; __LOOP__0)

__LOOP__1 =
  ((k?PENDING.ptTimeLeft ;
    ((__LOOP__2 [> (k?ABORT._ ; ([__exit__PENDING_DONE:0,0] ; idle)))
   [> (__gx__PENDING_DONE -> [__null__:0,0]))) ; __LOOP__1)

__LOOP__2 =
  ((((__result__pendingExpired:1000,1250] ;
    ((__gf__1 -> (k!PENDING_TIME.ptTimeLeft ; ([__exit__PENDING_DONE:0,0] ; idle))) +
     (__gf__0 -> k!PENDING_TIME.ptTimeLeft))) ; idle)
   [> [__timer__:1000000,1000000]) ; __LOOP__2)

__LOOP__3 = ((k?ALERT._ ; (__LOOP__4 [> (k?ABORT._ ; [linkLEDOff:1000,1250]))) ; __LOOP__3)
__LOOP__4 = ((([linkLEDToggle:1000,1250] ; idle) [> [__timer__:250000,250000]) ; __LOOP__4)
__LOOP__5 = ((k?CONTROL.displayMsg ; [Display:10000,20250]) ; __LOOP__5)
```

Net Representation


k!CONTROL_controlMsg^+1
k!CONTROL_displayMsg
k!ALERT_
k!ABORT_
k!PENDING_controlTimeLeft
k!PENDING_TIME_controlTimeLeft
k!PENDING_TIME_ptTimeLeft
k!PENDING_DONE
k!PENDING_controlTimeInterval^+1
k!ABORT_^+1
k!ALERT_^+1
[controlStatePendingTime:1000,1250]^+12
[initDisplay:1000,1250]^+5
[Display:10000,20250]^+5
[linkLEDOff:1000,1250]^+6
[linkLEDToggle:1000,1250]^+4
idle

A.4 Anti-lock Braking System

CANDLE Program

```
Control | Brake_0 | Brake_1 | Brake_2 | Brake_3 | Sensor_0 | Sensor_1 | Sensor_2 | Sensor_3
where

Control =
  every 16000 do
    snd(k, SPEED_REQ_0);
    rcv(k, SPEED_0, sSpeed_0);
    snd(k, SPEED_REQ_1);
    rcv(k, SPEED_1, sSpeed_1);
    snd(k, SPEED_REQ_2);
    rcv(k, SPEED_2, sSpeed_2);
    snd(k, SPEED_REQ_3);
    rcv(k, SPEED_3, sSpeed_3);

    case comput_acc()
      when 0 => null
      when 1 => ABS(); snd(k, PRESSURE, cPressure)
      when 2 => ASR(); snd(k, PRESSURE, cPressure)
    end case
  end every

Brake_0 =
  loop do
    rcv(k, PRESSURE, bPressure);
    adjustPressure_0()
  end loop

Brake_1 =
  loop do
    rcv(k, PRESSURE, bPressure);
    adjustPressure_1()
  end loop

Brake_2 =
  loop do
    rcv(k, PRESSURE, bPressure);
    adjustPressure_2()
  end loop

Brake_3 =
  loop do
    rcv(k, PRESSURE, bPressure);
    adjustPressure_3()
```

```
  end loop

Sensor_0 =
  loop do
    rcv(k, SPEED_REQ_0);
    readSensor_0();
    snd(k, SPEED_0, sSpeed_0)
  end loop

Sensor_1 =
  loop do
    rcv(k, SPEED_REQ_1);
    readSensor_1();
    snd(k, SPEED_1, sSpeed_1)
  end loop

Sensor_2 =
  loop do
    rcv(k, SPEED_REQ_2);
    readSensor_2();
    snd(k, SPEED_2, sSpeed_2)
  end loop

Sensor_3 =
  loop do
    rcv(k, SPEED_REQ_3);
    readSensor_3();
    snd(k, SPEED_3, sSpeed_3)
  end loop
```

Architecture Description

```
node control
  wordsize   : 32
  tickHz     : 1000
  processes  : Control
  scheduler  : COOPERATIVE
  ports      : CAN_0

process Control
  stacksize : 125
  channels  : k -> CAN_0

node w_0
  wordsize   : 32
  tickHz     : 1000
  processes  : Brake_0, Sensor_0
```

```
  scheduler  : COOPERATIVE
  ports      : CAN_0

process Brake_0
  stacksize : 25
  channels  : k -> CAN_0

process Sensor_0
  stacksize : 25
  channels  : k -> CAN_0

node w_1
  wordsize   : 32
  tickHz     : 1000
  processes  : Brake_1, Sensor_1
  scheduler  : COOPERATIVE
  ports      : CAN_0

process Brake_1
  stacksize : 25
  channels  : k -> CAN_0

process Sensor_1
  stacksize : 25
  channels  : k -> CAN_0

node w_2
  wordsize   : 32
  tickHz     : 1000
  processes  : Brake_2, Sensor_2
  scheduler  : COOPERATIVE
  ports      : CAN_0

process Brake_2
  stacksize : 25
  channels  : k -> CAN_0

process Sensor_2
  stacksize : 25
  channels  : k -> CAN_0

node w_3
  wordsize   : 32
  tickHz     : 1000
  processes  : Brake_3, Sensor_3
  scheduler  : COOPERATIVE
  ports      : CAN_0
```

```
process Brake_3
  stacksize : 25
  channels  : k -> CAN_0

process Sensor_3
  stacksize : 25
  channels  : k -> CAN_0

channel k
  bps : 100000
  messages : <PRESSURE:4, SPEED_REQ_0:0, SPEED_0:1, SPEED_REQ_1:0, SPEED_1:1,
                          SPEED_REQ_2:0, SPEED_2:1, SPEED_REQ_3:0, SPEED_3:1>
```

Computations Bounds

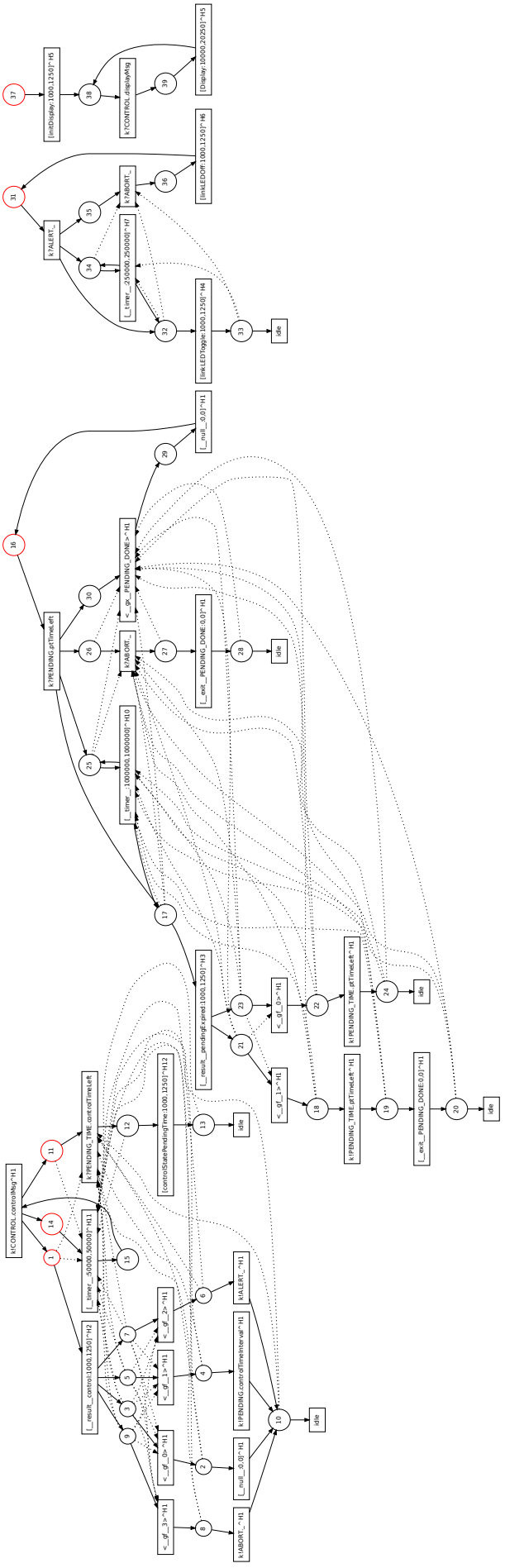
```
comput_acc          1000  1250
ABS                 1000  1250
ASR                 1000  1250
adjustPressure_0    1000  1250
adjustPressure_1    1000  1250
adjustPressure_2    1000  1250
adjustPressure_3    1000  1250
readSensor_0        1000  1250
readSensor_1        1000  1250
readSensor_2        1000  1250
readSensor_3        1000  1250
```

bCANDLE Model

```
(Control | (Brake_0 | (Brake_1 | (Brake_2 | (Brake_3 |
 (Sensor_0 | (Sensor_1 | (Sensor_2 | Sensor_3)))))))

where

Control =   __LOOP__0
Brake_0 =   __LOOP__1
Brake_1 =   __LOOP__2
Brake_2 =   __LOOP__3
Brake_3 =   __LOOP__4
Sensor_0 =  __LOOP__5
Sensor_1 =  __LOOP__6
Sensor_2 =  __LOOP__7
```

```
Sensor_3 =  __LOOP__8

__LOOP__0 =
  ((((k!SPEED_REQ_0._ ; (k?SPEED_0.sSpeed_0 ;
     (k!SPEED_REQ_1._ ; (k?SPEED_1.sSpeed_1 ;
     (k!SPEED_REQ_2._ ; (k?SPEED_2.sSpeed_2 ;
     (k!SPEED_REQ_3._ ; (k?SPEED_3.sSpeed_3 ;
     ([__result__comput_acc:1000,1250] ;
       ((__gf__0 -> [__null__:0,0]) +
       ((__gf__1 -> ([ABS:1000,1250] ; k!PRESSURE.cPressure)) +
        (__gf__2 -> ([ASR:1000,1250] ; k!PRESSURE.cPressure))))))))))) ; idle)
   [> [__timer__:16000,16000]) ; __LOOP__0)

__LOOP__1 = ((k?PRESSURE.bPressure ; [adjustPressure_0:1000,1250]) ; __LOOP__1)

__LOOP__2 = ((k?PRESSURE.bPressure ; [adjustPressure_1:1000,1250]) ; __LOOP__2)

__LOOP__3 = ((k?PRESSURE.bPressure ; [adjustPressure_2:1000,1250]) ; __LOOP__3)

__LOOP__4 = ((k?PRESSURE.bPressure ; [adjustPressure_3:1000,1250]) ; __LOOP__4)

__LOOP__5 = ((k?SPEED_REQ_0._ ; ([readSensor_0:1000,1250] ; k!SPEED_0.sSpeed_0)) ; __LOOP__5)

__LOOP__6 = ((k?SPEED_REQ_1._ ; ([readSensor_1:1000,1250] ; k!SPEED_1.sSpeed_1)) ; __LOOP__6)

__LOOP__7 = ((k?SPEED_REQ_2._ ; ([readSensor_2:1000,1250] ; k!SPEED_2.sSpeed_2)) ; __LOOP__7)

__LOOP__8 = ((k?SPEED_REQ_3._ ; ([readSensor_3:1000,1250] ; k!SPEED_3.sSpeed_3)) ; __LOOP__8)
```

Net Representation


k!CONTROL_controlMsg^*H1
k!PENDING_TIME_controlTimeLeft
[__timer__50000;50000]^*H1
[__result__control;1000;1250]^*H2
<__gf__0>^*H1
<__gf__1>^*H1
<__gf__2>^*H1
<__gf__3>^*H1
k!ALERT_^*H1
k!PENDING_controlTimeInterval^*H1
[__null__0;0]^*H1
k!ABORT_^*H1
[controlStatePendingTime;1000;1250]^*H12
idle
[__result__pendingExpired;1000;1250]^*H3
<__gf__0>^*H1
<__gf__1>^*H1
k!PENDING_TIME_ptTimeLeft^*H1
k!PENDING_TIME_ptTimeLeft^*H1
__exit__PENDING_DONE;0.0]^*H1
idle
k!PENDING_ptTimeLeft
[__timer__1000;1000;1000000]^*H10
k!ABORT_
<__gf__PENDING_DONE>^*H1
[__null__0;0]^*H1
__exit__PENDING_DONE;0.0]^*H1
idle
k!ALERT_
[__timer__250000;250000]^*H7
k!ABORT_
[linkLEDOff;1000;1250]^*H6
[linkLEDToggle;1000;1250]^*H4
idle
[initDisplay;1000;1250]^*H5
k!CONTROL_displayMsg
[Display;10000;20250]^*H5

B. UPPAAL MODELS

This section shows the generated UPPAAL models of the CAN communication and the flow regulator example respectively.

B.1 The CAN Communication Model

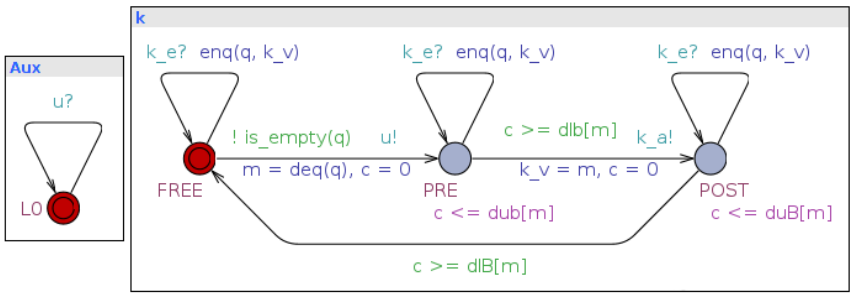


Fig. B.1: The UPPAAL model of the CAN communication.

B.2 Flow Regulator System Model

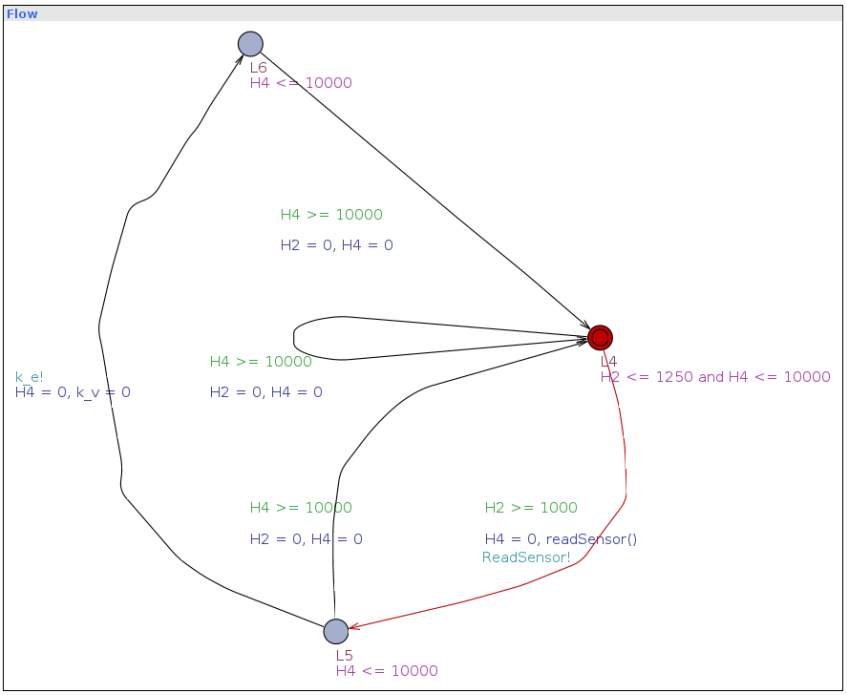


Fig. B.2: The UPPAAL model of the *Flow* process.

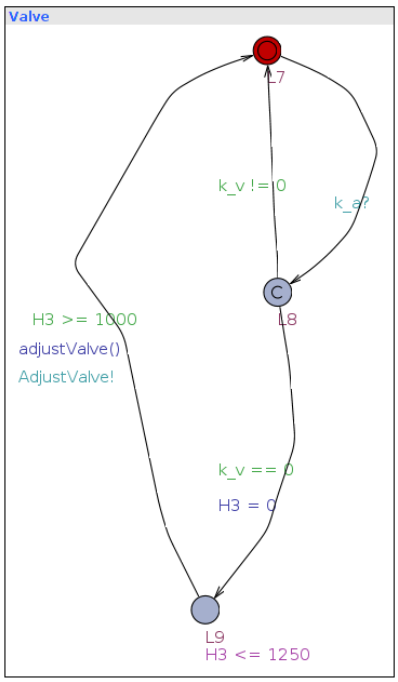


Fig. B.3: The UPPAAL model of the *Valve* process.

C. C SOURCE CODE

In this section, we present the C source code of the interrupt service routine (ISR), and the generated C code of the flow regulator example, respectively.

C.1 The C code of the ISR

```
1 /*
2  * file: bc.h
3  **/
4
5 #ifndef __BC_H
6 #define __BC_H
7
8 #include <stdbool.h>
9 #include <stdint.h>
10 #include <bcport.h>
11
12 enum bcGuardMasks {
13   BC_GUARD_EXCEPTION_FLAG          = 0x80000000U,
14   BC_GUARD_EXCEPTION_RESULT_MASK   = 0x7FFFFFFFU,
15 };
16
17 typedef uint32_t bcResult_t;
18
19 typedef uint32_t bcGuardSelect_t;
20
21 typedef void (* bcComputation_t)(void);
22
23 typedef struct bcNCB {
24   uint32_t *ptos;     /* pointer to top of stack; must be first
       field */
25   int32_t index;      /* index to transition for current
       computation */
26                       /* 0  : no current computation */
27                       /* -t : computation ongoing for transition
                            t */
28                       /* t  : computation completed for
                            transition t */
```

```
29   bcResult_t result; /* if the computation is a function, store
         the result here */
30 } bcNCB_t;
31
32
33 typedef struct bcPCB {
34   uint32_t index;     /* Index set as follows:
35                           0 if data is stale,
36                           1 if data is fresh and channel is external
37                           p if p is the place number of trigger of
                                sender transition
38                             and data is fresh
39                             and this is a LOCAL channel
40                        */
41   bcPortCanMessage_t message;
42 } bcPCB_t;
43
44
45 typedef enum bcAttributeType {
46   BC_TICK,
47   BC_IDLE,
48   BC_COMP,
49   BC_DELAY,
50   BC_GFUN,
51   BC_GEXN,
52   BC_GVAR,
53   BC_EXIT,
54   BC_SEND,
55   BC_RECV
56 } bcAttributeType_t;
57
58
59 /*
60  * Attributes should be assigned as follows:
61  *
62  * BC_TICK  - attribute should be 0; ignored
63  * BC_IDLE  - attribute should be 0; ignored
64  * BC_COMP  - attribute should be pointer to function for this
       computation
65  * BC_DELAY - attribute should be initial value of delay
66  * BC_GFUN - attribute should be 0 for negative guard, 1 for
       positive guard and between 2 and 255 for exception guard
67  * BC_GEXN - attribute should be 0 for negative guard, 1 for
       positive guard and between 2 and 255 for exception guard
68  * BC_GVAR - attribute should be 0 for negative guard, 1 for
       positive guard and between 2 and 255 for exception guard
69  * BC_SEND  - attribute should be address of a CAN message variable
```

```
70  * BC_RECV - attribute should be address of a CAN message variable
71  */
72 typedef uint32_t bcAttribute_t;
73
74
75 typedef struct bcTimer {
76   uint32_t index;     /* index to transition for current timer */
77                       /* 0 : timer not active */
78                       /* t : timer for transition t */
79   uint32_t value;     /* current value for active timer */
80                       /* undefined for inactive timer */
81 } bcTimer_t;
82
83
84 extern bcResult_t volatile bcExceptionResult;
85
86 void bcISR(void);
87 void bcRunSystem(void);
88 void bcNullComputation(void);
89 void bcIdleComputation(void);
90
91
92
93 #endif
```

```
1 /*
2  * file: bc.c
3  **/
4
5 #include <stdbool.h>
6 #include <assert.h>
7 #include <bc.h>
8 #include <bcport.h>
9 #include <bcgen.h>
10 #include <bcbitset.h>
11
12 bcResult_t volatile bcGuardResult = 0;
13
14 #ifdef BCGEN_CAN_REQUIRED
15
16 static bcPortCanMessage_t sendBuffer;
17 static uint32_t port;
18 static uint32_t len;
19 static uint32_t msgId;
20 static uint8_t *from;
21 static uint8_t *to;
22
23 #endif
```

```
24
25 static void react(uint32_t p);
26 static void fire(uint32_t p);
27 static void scheduleNextComputation(void);
28
29 void bcISR(void) {
30   int32_t i;
31   int32_t j;
32   bool stable = false;
33   bcGenPlaceSet_t lastMarking;
34
35   /* Just starting the ISR */
36   bcPortISREntryHook();
37
38 #ifdef BCGEN_TIMERS_REQUIRED
39
40   /* update soft timers */
41   for (i=0; i < BCGEN_N_TIMERS; i+=1) {
42     if (bcGenTimers[i].index != 0) {
                                          /* timer is active...     */
43       bcGenTimers[i].value -= 1;
                                               /* ...so decrement
           count */
44     }
45   }
46
47 #endif
48
49 #ifdef BCGEN_CAN_REQUIRED
50
51   /* update external port control blocks */
52   for (i=0; i < BCGEN_N_EXTERNAL_PORTS; i+=1) {
53     if (bcPortCanReady(i)) {
54       bcPortCanRead(i, &bcGenPCB[i].message);
55       bcGenPCB[i].index = 1;
56     }
57     else {
58       bcGenPCB[i].index = 0;
59     }
60   }
61
62   /* mark all local port control blocks as stale */
63   for (i=BCGEN_N_EXTERNAL_PORTS; i<BCGEN_N_PORTS; i+=1) {
64     bcGenPCB[i].index = 0;
65   }
66
67 #endif
```

```
68
69   /* react to marked places */
70 #if defined(BCGEN_DEBRUIJN)
71 {
72   /*
73    *  See C. Leiserson, H. Prokop, and K. Randall.
74    *      Using de Bruijn sequences to index a 1 in a computer
          word.
75    *      MIT Laboratory for Computer Science, 1998
76    *  for an explanation of this approach to identifying the bit
          number
77    *  of the rightmost 1 in a computer word. Used here to identify
          the
78    *  currently marked places during the reaction cycle.
          Preliminary
79    *  tests suggest that it's about 30% quicker than testing every
          bit
80    *  for this application.
81    *
82    *  It seems to slow the ISR by about 2.5% if you declare the
          following array as
83    *  const (and hence force it into flash).
84    */
85   static uint32_t debruijn[32] =
86     {0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20, 15, 25, 17, 4, 8,
87      31, 27, 13, 23, 21, 19, 16, 7, 26, 12, 18, 6, 11, 5, 10, 9};
88
89   uint32_t offset = 0;
90   uint32_t word = 0;
91   uint32_t rightmostBitVal = 0;
92   uint32_t place = 0;
93
94   bool firstIteration = true; // Used to detect the first
         iteration of while loop.
95
96   while (!stable) {
97     for (i = 0; i < BCGEN_N_PLACE_WORDS; i+=1) {
98       lastMarking[i] = bcGenMarked[i];
99     }
100    for (i = 0, offset = 0; i < BCGEN_N_PLACE_WORDS; i+=1,
           offset+=32) {
101      word = bcGenMarked[i];
102      while (word != 0) {
103        rightmostBitVal = (word & (-word));
104        word -= rightmostBitVal;
105        place = debruijn[((rightmostBitVal * 0x077CB531) >> 27)] +
               offset;
```

```
106           react(place);
107 #ifdef BCGEN_TIMERS_REQUIRED
108           for (j=0; j<BCGEN_N_TIMERS; j+=1) {
109             if (!(bitTest(bcGenMarked, bcGenTimers[j].index))) {
110               bcGenTimers[j].index = 0;
111             }
112           }
113 #endif
114         }
115       }
116       stable = true;
117       for (i = 0; i < BCGEN_N_PLACE_WORDS; i+=1) {
118         if (bcGenMarked[i] != lastMarking[i]) {
119           stable = false;
120           break;
121         }
122       }
123
124 #ifdef BCGEN_CAN_REQUIRED
125       /* mark all external port control blocks as stale after the
            first iteration*/
126       if(firstIteration == true){
127         for (i=0; i<BCGEN_N_EXTERNAL_PORTS; i+=1) {
128            bcGenPCB[i].index = 0;
129         }
130         firstIteration = false;
131       }
132
133       for (i=BCGEN_N_EXTERNAL_PORTS; i<BCGEN_N_PORTS; i+=1) {
134         bcGenPCB[i].index = 0;
135       }
136 #endif
137
138     }
139 }
140
141 #else
142     while (!stable) {
143       for (i = 0; i < BCGEN_N_PLACE_WORDS; i+=1) {
144         lastMarking[i] = bcGenMarked[i];
145       }
146       for (i=0; i < BCGEN_N_PLACES; i+=1) {
147         if (bitTest(bcGenMarked, i)) {
148           react(i);
149         }
150       }
151       stable = true;
```

```
152     for (i = 0; i < BCGEN_N_PLACE_WORDS; i+=1) {
153       if (bcGenMarked[i] != lastMarking[i]) {
154         stable = false;
155         break;
156       }
157     }
158   }
159 #endif
160
161   /* schedule next computation */
162   scheduleNextComputation();
163
164   /* About to leave the ISR */
165   bcPortISRExitHook();
166 }
167
168 static void react(uint32_t p) {
169   uint32_t tIndex = bcGenTransitions[p].index;
170
171   switch (bcGenTransitions[p].type) {
172   case BC_IDLE : {
173     bcGenNCB[tIndex].index = 0;          /* allow the scheduler to
          pass over this computation if it wants */
174     break;
175   }
176   case BC_COMP : {
177     if (bcGenNCB[tIndex].index < 0) {
178       /* computation has been scheduled but is not yet complete;
            nothing to do */
179     }
180     else if (bcGenNCB[tIndex].index == 0) { /* computation should
          be scheduled */
181       bcGenNCB[tIndex].index = -p;           /* show computation
            has been scheduled but is not completed */
182       bcGenNCB[tIndex].ptos = bcGenComputationInit(tIndex,
            (bcComputation_t)bcGenTransitions[p].attribute);
183     }
184     else { /* (bcGenNCB[tIndex].index > 0) so computation has been
          completed */
185       bcGenNCB[tIndex].index = 0;          /* ...so reset index to
            show computation is no longer active... */
186       fire(p);                              /* ...fire the
            transition */
187     }
188     break;
189   }
190   case BC_DELAY : {
```

```
191 #ifdef BCGEN_TIMERS_REQUIRED
192     if (bcGenTimers[tIndex].index != 0) {  /* timer is active */
193       if (bcGenTimers[tIndex].value == 0) { /* ... and has expired
              */
194         fire(p);
195         bcGenTimers[tIndex].index = 0;
196       }
197     }
198     else { /* start the timer */
199       bcGenTimers[tIndex].index = p;
200       bcGenTimers[tIndex].value =
            (uint32_t)bcGenTransitions[p].attribute;
201     }
202 #endif
203     break;
204   }
205   case BC_GFUN : {
206     if ((bcGuardSelect_t)bcGenNCB[tIndex].result ==
          (bcGuardSelect_t)bcGenTransitions[p].attribute) { /* this
          guard selected ... */
207       /* bcGuardResult = bcGenNCB[tIndex].result;

            save the result in case it's an exception */
208       fire(p);

            /* ... and fire transition */
209     }
210     break;
211   }
212   case BC_GVAR : {
213     if (((int32_t)bcGenTransitions[p].attribute == INT32_MAX) ||
          ((*(bcGuardSelect_t *)tIndex ==
          (bcGuardSelect_t)bcGenTransitions[p].attribute))) {
                      /* this guard selected ... */
214       fire(p);

            /* ... so fire transition */
215     }
216     break;
217   }
218   case BC_GEXN : {
219 #ifdef BCGEN_EXN_REQUIRED
220     if (bcGenECB[tIndex] & (1 <<
          (uint32_t)bcGenTransitions[p].attribute)) {
                                 /* exception is raised ...      */
221       bcGenECB[tIndex] &= ~(1 <<
            (uint32_t)bcGenTransitions[p].attribute);
```

```
                                        /* ... clear the exception ...
            */
222         fire(p);

            /* ... and fire transition      */
223       }
224 #endif
225       break;
226     }
227     case BC_EXIT : {
228 #ifdef BCGEN_EXN_REQUIRED
229       bcGenECB[tIndex] |= (1 <<
            (uint32_t)bcGenTransitions[p].attribute);
                                         /* raise the exception      */
230       fire(p);

            /* ... and fire transition */
231 #endif
232       break;
233     }
234     case BC_SEND : {
235 #ifdef BCGEN_CAN_REQUIRED
236       port = (tIndex & 0x00000007);
237       len = ((tIndex >> BCGEN_MSG_LEN_OFFSET) & 0x0000000F);
238       msgId = tIndex >> BCGEN_MSG_ID_OFFSET;
239       from = (uint8_t *)bcGenTransitions[p].attribute;
240
241       if ((BCGEN_N_EXTERNAL_PORTS > 0) && (port <
            BCGEN_N_EXTERNAL_PORTS)) {
242         uint32_t i;
243         to = (uint8_t *)&sendBuffer.dataA;
244         for (i=len; i != 0; i--) {
245           *to++ = *from++;
246         }
247         sendBuffer.id = msgId;
248         sendBuffer.len = len;
249         bcPortCanWrite(port, &sendBuffer);
250         fire(p);
251       }
252       else {
253         uint32_t i;
254         if (bcGenPCB[port].index == 0) {
255           bcGenPCB[port].index = p;
256           to = (uint8_t *)&bcGenPCB[port].message.dataA;
257           for (i=len; i != 0; i--) {
258             *to++ = *from++;
259           }
```

```
260           bcGenPCB[port].message.id = msgId;
261           bcGenPCB[port].message.len = len;
262           fire(p);
263         }
264       }
265 #endif
266       break;
267     }
268     case BC_RECV : {
269 #ifdef BCGEN_CAN_REQUIRED
270       port = (tIndex & 0x00000007);
271       msgId = tIndex >> BCGEN_MSG_ID_OFFSET;
272       if ((bcGenPCB[port].index != 0)  &&
                                                    /* fresh
           data available                          */
273           (bcGenPCB[port].message.id == msgId)) {
                                                    /* and
               matching id                         */
274         uint32_t i;
275         len = ((tIndex >> BCGEN_MSG_LEN_OFFSET) & 0x0000000F);
276         assert(len == bcGenPCB[port].message.len);
                                                /* check that
             we've got the right number of bytes */
277         to =   (uint8_t *)bcGenTransitions[p].attribute;
278         from = (uint8_t *)&bcGenPCB[port].message.dataA;
279         for (i=len; i != 0; i--) {

             /* copy to the user data variable     */
280           *to++ = *from++;
281         }
282         fire(p);

             /* and fire the transition            */
283       }
284 #endif
285       break;
286     }
287     default:
288       assert(false);        /* should not happen */
289     }
290 }
291
292 static void fire(uint32_t p) {
293   bcGenTransition_t t = bcGenTransitions[p];
294   uint32_t i;
295
296   bitClear(bcGenMarked, p);
```

```
297   for (i = 0; i < BCGEN_N_PLACE_WORDS; i++) {
298     bcGenMarked[i] &= ~(t.vulnerable[i]);
299     bcGenMarked[i] |= t.target[i];
300   }
301 }
302
303
304 static uint32_t scheduleNextNCBindex = 0;
305
306 #if defined(BCGEN_SCHEDULE_ROUND_ROBIN)
307 static void scheduleNextComputation(void) {
308     if (++scheduleNextNCBindex == BCGEN_N_NETS) {
309       scheduleNextNCBindex = 1;
310     }
311
312     if (bcGenNCB[scheduleNextNCBindex].index != 0) { /* there is a
          computation to consider */
313       if (bitTest(bcGenMarked,
            -(bcGenNCB[scheduleNextNCBindex].index))) {
314         bcGenCurrentNCBPtr = &bcGenNCB[scheduleNextNCBindex]; /*
              trigger still marked so schedule this computation */
315       }
316       else {
317         bcGenNCB[scheduleNextNCBindex].index = 0;       /* trigger
              no longer marked; abandon this computation */
318         bcGenCurrentNCBPtr = &bcGenNCB[0];              /* and
              schedule the idle computation */
319       }
320     }
321     else {
322       bcGenCurrentNCBPtr = &bcGenNCB[0];                /* no
          computation available here; schedule the idle computation
          */
323     }
324 }
325
326 #elif defined(BCGEN_SCHEDULE_FIXED_PRIORITY)
327
328 static void scheduleNextComputation(void) {
329     scheduleNextNCBindex = 0;
330     while (++scheduleNextNCBindex < BCGEN_N_NETS) {
331       if (bcGenNCB[scheduleNextNCBindex].index != 0) {
332         if (bitTest(bcGenMarked,
              -(bcGenNCB[scheduleNextNCBindex].index))) {
333           bcGenCurrentNCBPtr = &bcGenNCB[scheduleNextNCBindex]; /*
                trigger still marked so schedule this computation */
334           return;
```

```
335         }
336         else {
337           bcGenNCB[scheduleNextNCBindex].index = 0;       /*
                trigger no longer marked; abandon this computation */
338         }
339       }
340     }
341     bcGenCurrentNCBPtr = &bcGenNCB[0];                  /* no
          computation available; schedule the idle computation */
342 }
343
344 #elif defined(BCGEN_SCHEDULE_COOPERATIVE)
345
346 static bool coroutineRunning = false;
347
348 void bcCoroutine(void) {
349   bcNCB_t volatile *savedNCBPtr = bcGenCurrentNCBPtr;
350   int32_t tIndex = 0;
351   int32_t index = 0;
352
353   coroutineRunning = true;
354   while (++scheduleNextNCBindex < BCGEN_N_NETS) {
355     tIndex = -(bcGenNCB[scheduleNextNCBindex].index);
356     if (tIndex != 0) {
          /* there is a computation to consider */
357       if (bitTest(bcGenMarked, tIndex)) {
358         bcGenCurrentNCBPtr = &bcGenNCB[scheduleNextNCBindex];
              /* trigger still marked so ... */
359         ((bcComputation_t)bcGenTransitions[tIndex].attribute)();
              /* ... run the computation      */
360         index = bcGenCurrentNCBPtr->index;
361         bcGenCurrentNCBPtr->index = -index;
362       }
363       else {
364         bcGenNCB[scheduleNextNCBindex].index = 0;
              /* trigger no longer marked; abandon this computation */
365       }
366     }
367   }
368   bcGenCurrentNCBPtr = savedNCBPtr;
        /* no more computations available; return */
369   coroutineRunning = false;
370 }
371
372 bool bcExecutingAsCoroutine(void) {
373   return coroutineRunning;
374 }
```

```
375
376
377 static void scheduleNextComputation(void) {
378     scheduleNextNCBindex = 0;
379     while (++scheduleNextNCBindex < BCGEN_N_NETS) {
380       if (bcGenNCB[scheduleNextNCBindex].index != 0) {
381         if (bitTest(bcGenMarked,
                -(bcGenNCB[scheduleNextNCBindex].index))) {
382           bcGenCurrentNCBPtr = &bcGenNCB[scheduleNextNCBindex]; /*
                trigger still marked so schedule this computation */
383           return;
384         }
385         else {
386           bcGenNCB[scheduleNextNCBindex].index = 0;       /*
                trigger no longer marked; abandon this computation */
387         }
388       }
389     }
390     bcGenCurrentNCBPtr = &bcGenNCB[0];                  /* no
          computation available; schedule the idle computation */
391 }
392
393 #else
394
395 #error : No scheduling policy defined
396
397 #endif
398
399
400 void bcRunSystem(void) {
401   bcPortBSPinit();
402   bcGenStackInit();
403   bcPortTimerInit();
404   bcPortStartSystem();
405 }
406
407
408 void bcNullComputation(void) {
409   int32_t index = bcGenCurrentNCBPtr->index;
410   bcGenCurrentNCBPtr->index = -index;
411   bcIdleComputation();
412 }
413
414
415 void bcIdleComputation(void) {
416   while (true) {
417   }
```

```
418 }
```

C.2 The C code of the flow node

```
 1 /*
 2  * File: bcgen.h of flow node.
 3  */
 4
 5 #ifndef __BCGEN_H
 6 #define __BCGEN_H
 7
 8 #include <bcport.h>
 9 #include <bc.h>
10
11 /*
12  * Define the tick rate for the ISR
13  */
14 #define BCGEN_TICK_HZ 1000
15
16 /*
17  * Define BCGEN_TIMERS_REQUIRED to include code for use
18  * of soft timers
19  */
20 #define BCGEN_TIMERS_REQUIRED
21
22
23 /*
24  * Define BCGEN_CAN_REQUIRED to include code for communication
25  * between processes either via CAN or locally
26  */
27 #define BCGEN_CAN_REQUIRED
28
29
30
31 /*
32  * Define BCGEN_DEBRUIJN to choose fast iteration through marked
      places in the
33  * main loop in the ISR
34  */
35 #define BCGEN_DEBRUIJN
36
37
38 /* Must define one of BCGEN_SCHEDULE_ROUND_ROBIN or
      BCGEN_SCHEDULE_FIXED_PRIORITY
39  * or BCGEN_SCHEDULE_COOPERATIVE or BCGEN_SCHEDULE_HYBRID
40  */
41 #define BCGEN_SCHEDULE_ROUND_ROBIN
```

```
42
43 #if defined(BCGEN_SCHEDULE_COOPERATIVE) ||
       defined(BCGEN_SCHEDULE_HYBRID)
44 void bcCoroutine(void);
45 bool bcPrimaryCoopCall(void);
46 bool bcSecondaryCoopCall(void);
47 #endif
48
49 enum {
50   BCGEN_N_NETS              = 2,
51   BCGEN_N_PLACES            = 4,
52   BCGEN_N_PLACE_WORDS       = 1,
53   BCGEN_N_TIMERS            = 1,
54   BCGEN_N_PORTS             = 1,
55   BCGEN_N_EXTERNAL_PORTS    = 1,
56   BCGEN_MSG_ID_OFFSET       = 7,
57   BCGEN_MSG_LEN_OFFSET      = 3,
58   BCGEN_N_STACK_WORDS       = 40
59 };
60
61 typedef bcPortWord_t bcGenPlaceSet_t[BCGEN_N_PLACE_WORDS];
62
63 /*
64  * The index field in bcGenTransition_t below has a variety of
        uses.
65  * It depends on the attribute type and is set as follows:
66  *
67  * BC_TICK  - index should be 0; ignored
68  * BC_IDLE  - index should be 0; ignored
69  * BC_COMP  - index should be index of the NCB that owns this
        computation
70  * BC_DELAY - index should be index of the soft timer tracking
        this delay
71  * BC_GUARD - index should be index of the NCB that owns this guard
72  * BC_SEND  - index should be ((message id << id_offset) |
        (message len << len_offset) | pcb id)
73  * BC_RECV  - index should be as for BC_SEND
74  *
75  *
76  * attribute should be set as defined in bc.h
77  */
78 typedef struct bcGenTransition {
79   bcAttributeType_t type;
80   uint32_t index;
81   bcGenPlaceSet_t vulnerable;
82   bcAttribute_t attribute;
83   bcGenPlaceSet_t target;
```

```
84 } bcGenTransition_t;
85
86
87 extern bcNCB_t volatile bcGenNCB[BCGEN_N_NETS];
88 extern bcNCB_t volatile * volatile bcGenCurrentNCBPtr;
89 extern bcTimer_t volatile bcGenTimers[BCGEN_N_TIMERS];
90 extern bcPCB_t bcGenPCB[BCGEN_N_PORTS];
91 extern bcGenPlaceSet_t bcGenMarked;
92 extern bcGenTransition_t const bcGenTransitions[BCGEN_N_PLACES];
93
94 void bcGenStackInit(void);
95 uint32_t* bcGenComputationInit(uint32_t net, bcComputation_t comp);
96
97 /*****************  User procedure and function prototypes
       **********/
98
99 void bcGenCompT2(void);
100
101 #endif
```

```
1 /*
2  * File: bcgen.c of flow node.
3  */
4
5 #include <stdint.h>
6 #include <bc.h>
7 #include <bcport.h>
8 #include <bcgen.h>
9 #include <user.h>
10
11 static void computationDone(void);
12
13 static uint32_t stack[BCGEN_N_STACK_WORDS];
14 static uint32_t const stackBaseIndex[BCGEN_N_NETS] = {39, 19};
15
16 bcNCB_t volatile bcGenNCB[BCGEN_N_NETS] = {
17   {(uint32_t *)0, 0, (bcResult_t)0},
18   {(uint32_t *)0, 0, (bcResult_t)0}
19 };
20
21 bcNCB_t volatile * volatile bcGenCurrentNCBPtr = &bcGenNCB[0];
22 bcPortWord_t bcGenMarked[BCGEN_N_PLACE_WORDS] = {
23   0x0000000CU
24 };
25
26 bcGenTransition_t const bcGenTransitions[BCGEN_N_PLACES] = {
27   {BC_IDLE, 0, {0x00000000U}, (bcAttribute_t)bcIdleComputation,
       {0x00000000U}},
```

```
28   {BC_SEND, (uint32_t)((0 << BCGEN_MSG_ID_OFFSET) | (sizeof(fFlow)
         << BCGEN_MSG_LEN_OFFSET) | 0), {0x00000000U},
         (bcAttribute_t)&fFlow, {0x00000000U}},
29   {BC_COMP, 1, {0x00000000U}, (bcAttribute_t)bcGenCompT2,
         {0x00000002U}},
30   {BC_DELAY, 0, {0x00000006U}, (bcAttribute_t)10, {0x0000000CU}}
31 };
32
33 bcTimer_t volatile bcGenTimers[BCGEN_N_TIMERS] = {
34   {0U, 0U}
35 };
36
37 bcPCB_t bcGenPCB[BCGEN_N_PORTS] = {
38   {0U, {0U, 0U, 0U, 0U}}
39 };
40
41 void bcGenStackInit(void) {
42   uint32_t i;
43
44   for (i=0; i<BCGEN_N_NETS; i+=1) {
45     if (i > 0) {
46       /* Put some stack markers in to help debugging */
47       stack[stackBaseIndex[i]+1] = 0xFEEDFACE;
48     }
49     bcGenNCB[i].ptos =
50       bcPortComputationInit(&stack[stackBaseIndex[i]],
51                             bcIdleComputation);
52   }
53   stack[0] = 0xFEEDFACE;
54 }
55
56 uint32_t * bcGenComputationInit(uint32_t net, bcComputation_t
       comp) {
57   return bcPortComputationInit(&stack[stackBaseIndex[net]], comp);
58 }
59
60 /***************************** User procedure calls
       ******************/
61
62 void bcGenCompT2(void) {
63   readSensor();
64   computationDone();
65 }
66
67 /***************************** Local functions
       ************************/
68
```

```
69 static void computationDone(void) {
70   int32_t index = bcGenCurrentNCBPtr->index;
71
72   bcGenCurrentNCBPtr->index = -index;
73 #if defined(BCGEN_SCHEDULE_COOPERATIVE) ||
       defined(BCGEN_SCHEDULE_HYBRID)
74   if (bcPrimaryCoopCall()) {
75     bcCoroutine();
76   }
77   else if (bcSecondaryCoopCall()) {
78     return;
79   }
80 #endif
81   bcIdleComputation();
82 }
```

C.3 The C code of the valve node

```
1 /*
2  * File: bcgen.h of valve node.
3  */
4
5 #ifndef __BCGEN_H
6 #define __BCGEN_H
7
8 #include <bcport.h>
9 #include <bc.h>
10
11 /*
12  * Define the tick rate for the ISR
13  */
14 #define BCGEN_TICK_HZ 1000
15
16 /*
17  * Define BCGEN_TIMERS_REQUIRED to include code for use
18  * of soft timers
19  */
20
21 /*
22  * Define BCGEN_CAN_REQUIRED to include code for communication
23  * between processes either via CAN or locally
24  */
25 #define BCGEN_CAN_REQUIRED
26
27
28
29 /*
```

```
30  * Define BCGEN_DEBRUIJN to choose fast iteration through marked
        places in the
31  * main loop in the ISR
32  */
33 #define BCGEN_DEBRUIJN
34
35
36 /* Must define one of BCGEN_SCHEDULE_ROUND_ROBIN or
       BCGEN_SCHEDULE_FIXED_PRIORITY
37  * or BCGEN_SCHEDULE_COOPERATIVE or BCGEN_SCHEDULE_HYBRID
38  */
39 #define BCGEN_SCHEDULE_ROUND_ROBIN
40
41 #if defined(BCGEN_SCHEDULE_COOPERATIVE) ||
       defined(BCGEN_SCHEDULE_HYBRID)
42 void bcCoroutine(void);
43 bool bcPrimaryCoopCall(void);
44 bool bcSecondaryCoopCall(void);
45 #endif
46
47 enum {
48   BCGEN_N_NETS              = 2,
49   BCGEN_N_PLACES            = 3,
50   BCGEN_N_PLACE_WORDS       = 1,
51   BCGEN_N_PORTS             = 1,
52   BCGEN_N_EXTERNAL_PORTS    = 1,
53   BCGEN_MSG_ID_OFFSET       = 7,
54   BCGEN_MSG_LEN_OFFSET      = 3,
55   BCGEN_N_STACK_WORDS       = 40
56 };
57
58 typedef bcPortWord_t bcGenPlaceSet_t[BCGEN_N_PLACE_WORDS];
59
60 /*
61  * The index field in bcGenTransition_t below has a variety of
        uses.
62  * It depends on the attribute type and is set as follows:
63  *
64  * BC_TICK  - index should be 0; ignored
65  * BC_IDLE  - index should be 0; ignored
66  * BC_COMP  - index should be index of the NCB that owns this
        computation
67  * BC_DELAY - index should be index of the soft timer tracking
        this delay
68  * BC_GUARD - index should be index of the NCB that owns this guard
69  * BC_SEND  - index should be ((message id << id_offset) |
        (message len << len_offset) | pcb id)
```

```
70  * BC_RECV - index should be as for BC_SEND
71  *
72  *
73  * attribute should be set as defined in bc.h
74  */
75 typedef struct bcGenTransition {
76   bcAttributeType_t type;
77   uint32_t index;
78   bcGenPlaceSet_t vulnerable;
79   bcAttribute_t attribute;
80   bcGenPlaceSet_t target;
81 } bcGenTransition_t;
82
83
84 extern bcNCB_t volatile bcGenNCB[BCGEN_N_NETS];
85 extern bcNCB_t volatile * volatile bcGenCurrentNCBPtr;
86 extern bcPCB_t bcGenPCB[BCGEN_N_PORTS];
87 extern bcGenPlaceSet_t bcGenMarked;
88 extern bcGenTransition_t const bcGenTransitions[BCGEN_N_PLACES];
89
90 void bcGenStackInit(void);
91 uint32_t* bcGenComputationInit(uint32_t net, bcComputation_t comp);
92
93 /***************** User procedure and function prototypes
      **********/
94
95 void bcGenCompT2(void);
96
97 #endif
```

```
1 /*
2  * File: bcgen.c of valve node.
3  */
4
5 #include <stdint.h>
6 #include <bc.h>
7 #include <bcport.h>
8 #include <bcgen.h>
9 #include <user.h>
10
11 static void computationDone(void);
12
13 static uint32_t stack[BCGEN_N_STACK_WORDS];
14 static uint32_t const stackBaseIndex[BCGEN_N_NETS] = {39, 19};
15
16 bcNCB_t volatile bcGenNCB[BCGEN_N_NETS] = {
17   {(uint32_t *)0, 0, (bcResult_t)0},
18   {(uint32_t *)0, 0, (bcResult_t)0}
```

```
19 };
20
21 bcNCB_t volatile * volatile bcGenCurrentNCBPtr = &bcGenNCB[0];
22 bcPortWord_t bcGenMarked[BCGEN_N_PLACE_WORDS] = {
23   0x00000002U
24 };
25
26 bcGenTransition_t const bcGenTransitions[BCGEN_N_PLACES] = {
27   {BC_IDLE, 0, {0x00000000U}, (bcAttribute_t)bcIdleComputation,
       {0x00000000U}},
28   {BC_RECV, (uint32_t)((0 << BCGEN_MSG_ID_OFFSET) | (sizeof(vFlow)
       << BCGEN_MSG_LEN_OFFSET) | 0), {0x00000000U},
       (bcAttribute_t)&vFlow, {0x00000004U}},
29   {BC_COMP, 1, {0x00000000U}, (bcAttribute_t)bcGenCompT2,
       {0x00000002U}}
30 };
31
32 bcPCB_t bcGenPCB[BCGEN_N_PORTS] = {
33   {0U, {0U, 0U, 0U, 0U}}
34 };
35
36 void bcGenStackInit(void) {
37   uint32_t i;
38
39   for (i=0; i<BCGEN_N_NETS; i+=1) {
40     if (i > 0) {
41       /* Put some stack markers in to help debugging */
42       stack[stackBaseIndex[i]+1] = 0xFEEDFACE;
43     }
44     bcGenNCB[i].ptos =
45       bcPortComputationInit(&stack[stackBaseIndex[i]],
46                             bcIdleComputation);
47   }
48   stack[0] = 0xFEEDFACE;
49 }
50
51 uint32_t * bcGenComputationInit(uint32_t net, bcComputation_t
     comp) {
52   return bcPortComputationInit(&stack[stackBaseIndex[net]], comp);
53 }
54
55 /***************************** User procedure calls
     *****************/
56
57 void bcGenCompT2(void) {
58   adjustValve();
59   computationDone();
```

```
60 }
61
62 /*************************** Local functions
       ************************/
63
64 static void computationDone(void) {
65   int32_t index = bcGenCurrentNCBPtr->index;
66
67   bcGenCurrentNCBPtr->index = -index;
68 #if defined(BCGEN_SCHEDULE_COOPERATIVE) ||
       defined(BCGEN_SCHEDULE_HYBRID)
69   if (bcPrimaryCoopCall()) {
70     bcCoroutine();
71   }
72   else if (bcSecondaryCoopCall()) {
73     return;
74   }
75 #endif
76   bcIdleComputation();
77 }
```

BIBLIOGRAPHY

Abrial, J.-R., Börger, E., and Langmaack, H., editors (1996). *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes In Computer Science*, London, UK. Springer-Verlag.

Absint (2012). Home page of aiT tool. http://www.absint.com/ait/.

Aceto, L., Bouyer, P., Burgueño, A., and Larsen, K. G. (1998). The Power of Reachability Testing for Timed Automata. In *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 245–256, London, UK. Springer-Verlag.

Aceto, L., Fokkink, W., and Verhoef, C. (2001). Chapter 3 - Structural Operational Semantics. In Bergstra, J., Ponse, A., and Smolka, S., editors, *Handbook of Process Algebra*, pages 197–292. Elsevier Science.

Alber, A. (2004). Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. In *Embedded World*, pages 235–252, Nürnberg, Germany.

Alur, R. and Dill, D. (1994). A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235.

Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., and Yi, W. (2003). TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *Procedings of FORMATS03, number 2791 in LNCS*, pages 60–72. Springer-Verlag.

Armengaud, E., Tengg, A., Driussi, M., Karner, M., Steger, C., and Weiss, R. (2009). Automotive software architecture: Migration challenges from an event-triggered to a time-triggered communication scheme. In *Intelligent solutions in Embedded Systems, 2009 Seventh Workshop on*, pages 95 –103.

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

Baronti, P., Pillai, P., Chook, V., Chessa, S., Gotta, A., and Fun-Hu, Y. (2007). Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. *Computer Communications*, 30(7):1655–1695.

Barry, R. (2009). *Using the FreeRTOS Real Time Kernel: A Practical Guide*. Real Time Engineers Limited.

Bedin, R., Blazy, S., Favre-Felix, D., Leroy, X., Pantel, M., and Souyris, J. (2012). Formally verified optimizing compilation in ACG-based flight control software. In *Embedded Real Time Software and Systems (ERTS 2012)*.

Behrmann, G., David, A., and Larsen, K. (2004). A Tutorial on Uppaal. In *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT04), number 3185 in LNCS*. Springer-Verlag.

Bengtsson, J. (2012). Memtime Tool. [Online]. Available: http://www.update.uu.se/ johanb/memtime/.

Benveniste, A. and Berry, G. (1991). The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270 –1282.

Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. (2003). The synchronous languages 12 years later. In *Proceedings of The IEEE*, volume 91, pages 64–83.

Berry, G. (2000). *The foundations of Esterel*, pages 425–454. MIT Press, Cambridge, MA, USA.

Berry, G., Ramesh, S., and Shyamasundar, R. K. (1993). Communicating reactive processes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 85–98, New York, NY, USA. ACM.

Berthomieu, B. and Diaz, M. (1991). Modeling and verification of time dependent systems using time Petri nets. *Software Engineering, IEEE Transactions on*, 17(3):259 –273.

Bertin, V., Closse, E., Poize, M., Pulou, J., Sifakis, J., Venier, P., Weil, D., and Yovine, S. (2001). Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. In *Proceedings of 40th Conference on Decision and Control, CDC'01*, pages 2875–2880.

Bertin, V., Poize, M., Pulou, J., and Sifakis, J. (2000). Towards validated real-time software. In *Proceedings of the 12th Euromicro Conference of Real Time Systems*, pages 157–164.

Blazy, S. (2008). *Sémantiques formelles*. Habilitation à diriger les recherches, Université Évry Val d'Essone.

Blazy, S. and Leroy, X. (2009). Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288.

Bosch, R. (1991). CAN specification version 2.0. Bosch GmbH.

Bosch GmbH, R. (2011). *Automotive Handbook*. Wiley-Blackwell, 8th edition.

Bradley, S. (1995). *An Implementable Formal Language for Hard Real-Time Systems*. PhD thesis, Department of Computing, University of Northumbria, UK.

Bradley, S., Henderson, W., Kendall, D., and Robson, A. (1994a). Application-Oriented Real-Time Algebra. *Software Engineering Journal*, 9(5):201–212.

Bradley, S., Henderson, W., Kendall, D., and Robson, A. (1994b). Designing and implementing correct real-time systems. In Langmaack, H., de Roever, W.-P., and Vytopil, J., editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT '94, Lubeck, Lecture Notes in Computer Science 863*, pages 228–246. Springer-Verlag.

Bradley, S., Henderson, W., Kendall, D., and Robson, A. (1994c). Practical formal development of real-time systems. In *11th IEEE Workshop on Real-Time Operating Systems and Software, RTOSS '94, Seattle*, pages 44–48.

Bradley, S., Henderson, W., Kendall, D., and Robson, A. (1996). Validation, verification and implementation of timed protocols using AORTA. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 205–220, London, UK, UK. Chapman & Hall, Ltd.

Brihaye, T., Bruyère, V., and Raskin, J. (2006). On model-checking timed automata with stopwatch observers. *Inf. Comput.*, 204(3):408–433.

Bril, R. J., Steffens, E. F. M., and Verhaegh, W. F. J. (2004). Best-case response times and jitter analysis of real-time tasks. *J. of Scheduling*, 7:133–147.

Brockway, M. (2010). *A Compositional Analysis of Broadcasting Embedded Systems*. PhD thesis, Northumbria University.

Burns, A. and Wellings, A. (2001). *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA.

Cascaval, C., Blundell, C., Michael, M., Cain, H., Wu, P., Chiras, S., and Chatterjee, S. (2008). Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):46–58.

Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., and Niebert, P. (2003). From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Languages, compilers, and tools for embedded systems*, pages 153–162, New York, NY, USA. ACM.

Caspi, P., Girault, A., and Pilaud, D. (1999). Automatic distribution of reactive systems for asynchronous networks of processors. *Software Engineering, IEEE Transactions on*, 25(3):416 –427.

Cassez, F. and Larsen, K. (2000). The impressive power of stopwatches. In Palamidessi, C., editor, *CONCUR 2000 Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 138–152. Springer Berlin / Heidelberg.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided Abstraction Refinement. In *International Conference on Computer Aided Verification (CAV'00)*.

Closse, E., Poize, M., Pulou, J., Sifakis, J., Venter, P., Weil, D., and Yovine, S. (2001). TAXYS: A Tool for the Development and Verification of Real-Time Embedded Systems. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 391–395.

Closse, E., Poize, M., Pulou, J., Venier, P., and Weil, D. (2002). SAXO-RT: Interpreting Esterel Semantics on a Sequential Execution Structure. *Electronic Notes in Theoretical Computer Science*, 65(5):80–94.

CompCert (2012). Homepage of CompCert project. http://compcert.inria.fr.

Dargaye, Z. (2009). *Vérification formelle d'un compilateur pour langages fonctionnels*. PhD thesis, Université Paris 7 Diderot.

Davis, R., Burns, A., Bril, R., and Lukkien, J. (2007). Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35:239–272.

Daws, C., Olivero, A., Tripakis, S., and Yovine, S. (1996). The tool KRONOS. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 208–219, Secaucus, NJ, USA. Springer-Verlag New York, Inc.

Day, T. and Roberts, S. (2002). A Simulation Model for Vehicle Braking Systems Fitted with ABS. SAE Technical Paper 2002-01-0559.

Dowson, M. (1997). The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84.

Ebner, C. (1998). Efficiency evaluation of a time-triggered architecture for vehicle body-electronics. In *Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on*, pages 62 –70.

Edwards, S. and Zeng, J. (2007). Code Generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, 2007:1–31.

Ellison, C. and Rosu, G. (2012). An executable formal semantics of C with applications. *SIGPLAN Not.*, 47(1):533–544.

Enoiu, E., Marinescu, R., Seceleanu, C., and Pettersson, P. (2012). ViTAL : A Verification Tool for EAST-ADL Models using UPPAAL PORT. In *Proceedings of the 17th IEEE International Conference on Engineering of Complex Computer Systems*, Paris, France.

Esterel-Tech (2005). *The Esterel v7 Reference Manual Version v7.30 . initial IEEE standardization proposal.* Esterel-Technologies, 679 av. Dr. J. Lefebvre 06270 Villeneuve- Loubet, France.

Eugenio, Y. (2008). CAN on parallel robots: How to control a stewart platform using CAN based motor controllers. In *Proceedings of the 12th iCC 2008*, Barcelona (Spain).

Feiler, P. H., Gluch, D., and Hudack, J. (2006). The Architecture Analysis & Design Language (AADL): An Introduction. No. CMU/SEI-2006-TN-011. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

Garavel, H., Lang, F., and Mateescu, R. (2002). Compiler construction using LOTOS NT. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 9–13, London, UK. Springer-Verlag.

Garavel, H., Lang, F., Mateescu, R., and Serwe, W. (2011). CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In Abdulla, P. and Leino, K., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin Heidelberg.

Garavel, H. and Serwe, W. (2006). State space reduction for process algebra specifications. *Theor. Comput. Sci.*, 351(2):131–145.

Garavel, H. and Sifakis, J. (1990). Compilation and verification of LOTOS specifications. In *Proceedings of the IFIP WG6.1 Tenth International Symposium on Protocol Specification, Testing and Verification X*, pages 379–394, Amsterdam, The Netherlands, The Netherlands. North-Holland Publishing Co.

Gendy, A. and Pont, M. (2008). Automatically Configuring Time-Triggered Schedulers for Use With Resource-Constrained, Single-Processor Embedded Systems. *Industrial Informatics, IEEE Transactions on*, 4(1):37–46.

Gluck, P. and Holzmann, G. (2002). Using Spin Model Checking for Flight Software Verification. In *Aerospace Conference Proceedings, 2002. IEEE.*

Goodenough, J. and Sha, L. (1988). The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. *Ada Lett.*, VIII(7):20–31.

Haberl, W., Tautschnig, M., and Baumgarten, U. (2008a). From COLA Models to Distributed Embedded Systems Code. *IAENG International Journal of Computer Science*, 35(3):427–437.

Haberl, W., Tautschnig, M., and Baumgarten, U. (2008b). Running COLA on Embedded Systems. In *Proceedings of The International MultiConference of Engineers and Computer Scientists 2008*, pages 922–928, Hong kong, China.

Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991). The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320.

Henderson, W., Kendall, D., Robson, A., and Bradley, S. (1998). χrma: An holistic approach to performance prediction of distributed real-time CAN systems. In *Proceedings of the 5th International CAN Conference (iCC'98), San Jose, California, USA*, pages 917–924. CiA.

Henzinger, T., Kopke, P., Puri, A., and Varaiya, P. (1995). What's decidable about hybrid automata? In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 373–382, New York, NY, USA. ACM.

Herber, P. (2010). *A Framework for Automated HW/SW Co-verification of SystemC Designs using Timed Automata*. PhD thesis, Elektrotechnik und Informatik, Technische Universität Berlin.

Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300.

Holzmann, G. (2012). Parallelizing the Spin Model Checker. In Donaldson, A. and Parker, D., editors, *Model Checking Software*, volume 7385 of *Lecture Notes in Computer Science*, pages 155–171. Springer Berlin Heidelberg.

Holzmann, G., Joshi, R., and Groce, A. (2011). Swarm Verification Techniques. *Software Engineering, IEEE Transactions on*, 37(6):845–857.

IAR-Systems (2012). Homepage of IAR visualSTATE state machine design tools. http://www.iar.com.

ISO-CAN (1993). ISO 11898-1 (1993) Road vehicles – Interchange of digital information – Controller Area Network (CAN) for high-speed communication. ISO Standard-11898, International Standards Organisation (ISO).

Jensen, H., Larsen, K., and Skou, A. (1996). Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL. In *Proceedings of the 2nd International Workshop on the SPIN Verification System*, pages 1–20.

Joseph, M. and Pandya, P. (1986). Finding Response Times in a Real-Time System. *Computer Journal*, 29(5):390–395.

Kendall, D. (2001a). CANDLE: A tool for efficient analysis of CAN control systems. In *Proceedings of the 1st Workshop on Real-Time Tools (RT-TOOLS'2001), Aalborg, Denmark*, Technical Report 2001-014, University of Uppsala.

Kendall, D. (2001b). *Formal Modelling and Analysis of Broadcasting Embedded Control Systems*. PhD thesis, Department of Computing Science, University of Newcastle upon Tyne, UK.

Kendall, D., Bradley, S., Henderson, W., and Robson, A. (1997). A formal basis for tool-supported simulation and verification of real-time CAN systems. In *Proceedings of 4th International CAN Conference (iCC'97)*, pages 719–727, Berlin. CAN in Automation.

Kendall, D., Bradley, S., Henderson, W., and Robson, A. (1998a). CANDLE: A high level language and development environment for high integrity CAN control systems. In *Proceedings of 4th IEE Workshop on Discrete Event Systems, Cagliari*, pages 58–63.

Kendall, D., Henderson, W., and Robson, A. (1998b). Modelling and analysis of broadcasting embedded control systems. In *IEE Colloquium on "Real-Time Systems: Can we meet future challenges?"*, Digest No. 1998/306, pages 8/1–4, London, UK. IEE.

Kesten, Y. and Pnueli, A. (1991). Timed and Hybrid Statecharts and Their Textual Representation. In *Proceedings of the Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 591–620, London, UK. Springer-Verlag.

Kopetz, H. (1991). Event-triggered versus time-triggered real-time systems. In Karshmer, A. and Nehmer, J., editors, *Operating Systems of the 90s and Beyond*, volume 563 of *Lecture Notes in Computer Science*, pages 86–101. Springer Berlin / Heidelberg.

Kopetz, H. (1995). *Predictably Dependable Computing Systems*, chapter The Time-Triggered Approach to Real-time System Design. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Kopetz, H. (1997). *Real-time systems : design principles for distributed embedded applications*. Kluwer international series in engineering and computer science. Boston ; London : Kluwer Academic.

Kugele, S., Tautschnig, M., Bauer, A., Schallhart, C., Merenda, S., Haberl, W., Kühnel, C., Müller, F., Wang, Z., Wild, D., Rittmann, S., and Wechs, M. (2007). COLA – The component language. Technical Report TUM-I0714, Institut für Informatik, Technische Universität München.

Labrosse, J. (2002). *MicroC/OS-II: The Real-Time Kernel*. CMP Books, 2nd edition.

Lee, E. A., Matic, S., Seshia, S. A., and Zou, J. (2009). The case for timing-centric distributed software (Invited Paper). In *Proceedings of the 2009 29th*

IEEE International Conference on Distributed Computing Systems Workshops, ICDCSW '09, pages 57–64, Washington, DC, USA. IEEE Computer Society.

LeGuernic, P., Gautier, T., Le Borgne, M., and Le Maire, C. (1991). Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321 –1336.

Léonard, L. and Leduc, G. (1997). An introduction to ET-LOTOS for the description of time-sensitive systems. *Comput. Netw. ISDN Syst.*, 29:271–292.

Leroy, X. (2009). A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446.

Lime, D., Roux, O., Seidner, C., and Traonouez, L.-M. (2009). Romeo: A Parametric Model-Checker for Petri Nets with Stopwatches. In Kowalewski, S. and Philippou, A., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, pages 54–57. Springer Berlin / Heidelberg.

Liu, C. and Layland, J. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of ACM*, 20(1):46–61.

Liu, I., Reineke, J., and Lee, E. A. (2010). A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties. In *44th Asilomar Conference on Signals, Systems, and Computers*, pages 2111–2115.

Liu, J. (2000). *Real-time systems*. Prentice Hall, Upper Saddle River, New Jersey.

Maaita, A. and Pont, M. (2005). Using "planned pre-emption" to reduce levels of task jitter in a time-triggered hybrid scheduler. In Koelmans, A., Bystrov, A., Pont, M., Ong, R., and Brown, A., editors, *Proceedings of the Second UK Embedded Forum*, pages 18–35, Birmingham, UK. University of Newcastle upon Tyne.

Mall, R. (2009). *Real-Time Systems: Theory and Practice*. Prentice Hall, 1st edition.

Mankin, J., Kaeli, D., and Ardini, J. (2009). Software transactional memory for multicore embedded systems. *SIGPLAN Not.*, 44(7):90–98.

MathWorks (2012). Simulink Reference Manual. http://www.mathworks.com.

MC68HC08 (2012). *MC68HC08AZ32/D datasheet*. MOTOROLA.

MISRA (2004). MISRA-C:2004 Guidelines for the use of the C language in critical systems.

Moreno, R., n Piedrafita, Salcedo, J., and Villarroel, L. (2006). Implementation of time Petri nets in real-time Java. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 178–187. ACM Press.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580.

Natale, M., Giusto, P., Zeng, H., and Ghosal, A. (2012). *Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice*. Springer.

Nicollin, X. and Sifakis, J. (1994). The algebra of timed processes, ATP: theory and application. *Inf. Comput.*, 114:131–178.

Nielson, H. and Nielson, F. (1991). *Semantics with Applications: A Formal Introduction*. Wiley professional computing. John Wiley and Sons.

Norstrom, C., Wall, A., and Yi, W. (1999). Timed automata as task models for event-driven systems. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 182 –189.

NXP (2009). *UM10211: LPC23xx User Manual, Rev. 03*. NXP Semiconductors.

Ortiz, M., Diaz, M., Bellido, F., Saez, E., and Quiles, F. (2011). Smart Home Automation Using Controller Area Network. In Abraham, A., Corchado, J., Gonzlez, S., and De Paz Santana, J., editors, *International Symposium on Distributed Computing and Artificial Intelligence*, volume 91 of *Advances in Intelligent and Soft Computing*, pages 167–174. Springer Berlin / Heidelberg.

Parent, M. and Cassin, M. (1999). CAN Bus in the Harsh and Hostile Oil Environment. In *Proceedings of the 6th iCC*, Turin (Italy).

Parr, T. (2013). StringTemplate Engine. http://www.stringtemplate.org/.

Petrović, D., Shahmirzadi, O., Ropars, T., Schiper, A., et al. (2012). Asynchronous Broadcast on the Intel SCC using Interrupts. In *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*, pages 24–29.

Plotkin, G. (2004). A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 6061:17 – 139.

Poledna, S., Mocken, T., Schiemann, J., and Beck, T. (1996). ERCOS: An Operating System for Automotive Applications. Technical report, SAE Technical Paper 960623.

Pont, M. (2008a). Applying time-triggered architectures in reliable embedded systems: challenges and solutions. *Elektrotechnik und Informationstechnik*, 125:401–405.

Pont, M. (2008b). *Patterns for time-triggered embedded systems*. ACM Press Books.

Pop, P., Eles, P., and Peng, Z. (2004). *Analysis and Synthesis of Distributed Real- time Embedded Systems*. Kluwer Academic.

Potop-Butucaru, D. and Caillaud, B. (2007). Correct-by-construction asynchronous implementation of modular synchronous specifications. *Fundamenta Informaticae*, 78:131–159.

Potop-Butucaru, D., Edwards, S., and Berry, G. (2007). *Compiling Esterel*. Springer-Verlag.

Rabbie, H. (2005). Implementing the LonTalk Protocol for Intelligent Distributed Control. In *Embedded System Conference, EIA Standard 709.1 Control Network Protocol Specification*.

Ratel, C., Halbwachs, N., and Raymond, P. (1991). Programming and verifying critical systems by means of the synchronous data-flow language LUSTRE. *SIGSOFT Softw. Eng. Notes*, 16:112–119.

Redell, O., El-khoury, J., and Törngren, M. (2004). The AIDA toolset for design and implementation analysis of distributed real-time control systems. *Microprocessors and Microsystems*, 28(4):163 – 182.

Redell, O. and Sanfridson, M. (2002). Exact best-case response time analysis of fixed priority scheduled tasks. In *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 165 – 172.

Riti, F. and Pozzi, F. (1999). CAN in Packaging and Plaster Machines. In *Proceedings of the 6th iCC*, Turin (Italy).

Samek, M. (2008). *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes.

Scarlett, J. and Brennan, R. (2006). Re-evaluating Event-Triggered and Time-Triggered Systems. In *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, pages 655 –661.

Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175 –1185.

Shavit, N. and Touitou, D. (1997). Software transactional memory. *Distributed Computing*, 10:99–116.

Sifakis, J., Tripakis, S., and Yovine, S. (2003). Building models of real-time systems from application software. *Proceedings of the IEEE*, 91(1):100 – 111.

Sloss, A., Symes, D., Rayfield, J., and Wright, C. (2004). *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann.

Texas-Instruments (2005). Texas instruments. TMS470R1x Controller Area Network (CAN) reference guide, literature number spnu197e.

Tidorum (2012). Home page of Bound-T tool. http://www.tidorum.fi/bound-t.

Tindell, K. and Burns, A. (1994). Guaranteed message latencies for distributed safety-critical hard real-time networks. Technical report, YCS 229, Department of Computer Science, University of York.

Tindell, K. and Hansson, H. (1995). Real Time Systems and Fixed Priority Scheduling. Technical report, Department of Computer Systems, Uppsala University.

Tindell, K., Hansson, H., and Wellings, A. (1994). Analysing real-time communications: controller area network (CAN). In *Proceedings of Real-Time Systems Symposium*, pages 259 –263.

Tovar, E. and Vasques, F. (1999). Real-time fieldbus communications using Profibus networks. *Industrial Electronics, IEEE Transactions on*, 46(6):1241–1251.

Tripakis, S. and Yovine, S. (2001). Timing Analysis and Code Generation of Vehicle Control Software using Taxys. In *Proceedings of Runtime Verification, RV'01*, pages 277–286.

Weil, D., Bertin, V., Closse, E., Poize, M., Venier, P., and Pulou, J. (2000). Efficient compilation of ESTEREL for real-time embedded systems. In *CASES*

'00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems, pages 2–8. ACM.

WindRiver (1999). *VxWorks programmer's guide 5.4*. Wind River Systems.

Xu, J. and Parnas, D. (2000). Priority Scheduling Versus Pre-Run-Time Scheduling. *The Journal of Real-Time Systems*, 18:7–23.

Yovine, S. (1993). *Méthodes et Outils pour la Vérification Symbolique de Systèmes Temporisés*. PhD thesis, Institut National Polytechnique de Grenoble.

Zhao, Y., Liu, J., and Lee, E. A. (2007). A programming model for time-synchronized distributed real-time systems. In *13th IEEE Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07*, pages 259 – 268.

Zhu, M. and Brooks, R. (2009). Comparison of Petri Net and Finite State Machine Discrete Event Control of Distributed Surveillance Networks. *International Journal of Distributed Sensor Networks*, 5(5):480–501.